MASTER EN SCIENCES INFORMATIQUES

Proposition d'une architecture orientée objet et distribuée d'emballage de bases de données relationnelles

Macours, Bernard

Award date:
2001

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Proposition d'une Architecture Orientée Objet et Distribuée d'Emballage de Bases de Données Relationnelles

BERNARD MACOURS

Mémoire présenté en vue de l'obtention du grade de
Licencié en Informatique (Horaire Décalé)

Année académique 2000-2001

Remerciements

Je remercie vivement Monsieur le Professeur Jean-Luc HAINAUT, qui a supervisé ce travail, pour sa disponibilité et ses conseils avisés. Merci à Monsieur Philippe THIRAN pour le temps qu'il a bien voulu me consacrer.

Je tiens également à remercier Sabine pour sa patience et son soutien, mes parents, frère et soeur pour leurs commentaires judicieux, leurs encouragements et le partage de leur propre expérience, ainsi que toutes les personnes qui, de près ou de loin, ont contribué à la réalisation de ce travail.

Table des Matières

Remerciements I

Table des Matières III

Préface V

I Introduction 1

I.1 Contexte général 1

I.2 Délimitation de la problématique 2

I.3 Critères d'évaluation 3
I.3.1 Structure des langages 3
I.3.2 Fonctionnalités particulières des langages 3

I.4 Base de données illustrative 4

II Le modèle JDBC 7

II.1 Architecture 7
II.1.1 Configuration logique 7
II.1.2 Configuration physique 8

II.2 Fonctionnement 8
II.2.1 Description du protocole 8
II.2.2 Exemple pratique 10

II.3 Évaluation du modèle 11
II.3.1 Structure des langages 11
II.3.2 Fonctionnalités particulières des langages 14
II.3.3 Résultat global 18

III Spécifications du modèle ARROW 19

III.1 Architecture 19
III.1.1 Middleware 19
III.1.2 Structure en deux couches 20
III.1.3 Répartition 21

III.2 Couche client 23
III.2.1 Manipulation des objets 23
III.2.2 Définition des objets 27

III.3 Couche Moteur 29

III.4 Limites et extensions potentielles 30
III.4.1 Gestion des données 30
III.4.2 Gestion de la répartition 30
III.4.3 Gestion des événements 30
III.4.4 Générateur d'objets 31

IV Langage de manipulation de ARROW 33

IV.1 Le noyau 33

IV.1.1 Coeur du noyau 33
IV.1.2 Extensions du noyau 34
IV.1.3 Modélisation de la base de données illustrative 35
IV.1.4 WProperty 38
IV.1.5 WObject 42
IV.1.6 WReference 45

IV.2 Collections 48
IV.2.1 WCondition 48
IV.2.2 WSortOrder 55
IV.2.3 WCollection 56

IV.3 Transactions 60
IV.3.1 WTransactionComponent 60
IV.3.2 WTransaction 62

IV.4 Limites et extensions potentielles 66

IV.5 Exemples applicatifs 67
IV.5.1 Exemples divers 67
IV.5.2 Gestion des commandes 68

V Langage de définition de ARROW 73

V.1 Définition du noyau 73
V.1.1 WProperty 74
V.1.2 WReference 84
V.1.3 WObject 86

V.2 Connexion au moteur 91

VI Moteur de ARROW 93

VI.1 Aperçu général 93

VI.2 Langage du moteur 94

VI.3 Implémentation JDBC/SQL 96

VII Evaluation de ARROW 99

VII.1 Evaluation du modèle 99
VII.1.1 Structure des langages 99
VII.1.2 Fonctionnalités particulières 100

VII.2 Tableau comparatif JDBC/ARROW 102

VII.3 Une autre approche 104
VII.3.1 Philosophie 104
VII.3.2 Architecture 105

VIII Conclusion 107

Bibliographie 109

Annexes 111

Package Java du modèle ARROW 111
Classes de base (package "arrow") 111
Spécialisations de WProperty (package "arrow.property") 116
Spécialisations de WReference (package "arrow.reference") 118
Spécialisations de WCondition (package "arrow.condition") 118
Moteur JDBC/SQL (package "arrow.sql") 121

Package Java de la base de données illustrative 123
Définition de la base de données (package "arrowx.database") 123

Préface

La cohabitation des paradigmes relationnel et objet au sein d'une même application est aujourd'hui monnaie courante. Typiquement, chacun d'eux se cantonne dans son domaine de prédilection: les bases de données pour le paradigme relationnel et les langages de programmation pour le paradigme objet.

Le rapprochement de ces deux univers peut s'envisager en intégrant des fonctionnalités orientées objets aux systèmes de gestion de bases de données relationnelles (SGBDR), ce que proposent certains vendeurs, ou en permettant aux langages de programmation orientés objets de s'interfacer aux SGBDR par l'intermédiaire d'un langage de requête tel que SQL. La mise en pratique de cette deuxième approche se concrétise par la présence d'un mélange de code orienté objet et de requêtes relationnelles dans les programmes.

Cette promiscuité dispose d'un certain nombre d'avantages, mais également d'inconvénients, que nous allons tenter de mettre en évidence en appuyant notre démarche sur le modèle JDBC (Java Database Connectivity), qui illustre parfaitement l'intégration de nos deux paradigmes (cfr II Le modèle JDBC). En réponse aux critiques développées, nous allons définir les spécifications d'un modèle alternatif qui constitue le coeur de ce travail: le modèle ARROW (cfr III Spécifications du modèle ARROW). Son rôle principal est de permettre aux développeurs d'applications de manipuler les SGBDR non plus au moyen des langages de requêtes de type SQL, mais bien via des objets.

Nous verrons que le modèle ARROW dispose d'un langage de manipulation des données, ou DML (Data Manipulation Language), et d'un langage de définition, ou DDL (Data Definition Language), dont la fonction est de décrire les structures de données objet exploitables par le DML (cfr IV Langage de manipulation de ARROW et V Langage de définition de ARROW). Nous étudierons également comment la communication avec les SGBDR est prise en charge et comment le DML est transformé en un langage compréhensible par les SGBDR (cfr VI Moteur de ARROW).

Nous terminerons par une évaluation du modèle ARROW, une confrontation avec le modèle JDBC et une étude comparative avec un ouvrage traitant la même problématique, mais sous un autre angle (VII Evaluation de ARROW).

I

Introduction

Avant d'aborder notre objectif principal, qui réside en la modélisation de ARROW (architecture distribuée et orientée objet d'encapsulation de bases de données relationnelles), nous allons, dans ce chapitre, introduire la problématique en présentant le contexte dans lequel est née l'idée de réaliser le présent travail.
Ce contexte va nous guider dans la définition d'un ensemble de critères d'évaluation auxquels seront soumis les modèles JDBC (cfr II.3) et ARROW (cfr VII.1)
En fin de chapitre, nous présenterons notre base de données illustrative qui servira de référence pour les exemples que nous développerons ensuite.

I.1 Contexte général

Nous nous positionnons en tant que service informatique d'une d'entreprise à la recherche d'une architecture permettant un accès distant à des données stockées dans une base relationnelle, au moyen d'un langage de type orienté-objet. L'objectif poursuivi est le développement de plusieurs nouvelles applications.
Pour ce faire, nous allons nous attarder non pas sur les aspects fonctionnels, mais bien sur les aspects technico-organisationnels caractérisés par les contraintes environnementales décrites ci-dessous. Ensemble, ces contraintes définissent le contexte d'investigation dans lequel nous nous inscrivons.

a) Environnement de réalisation

- **Base de données relationnelle.** La persistance des données est assurée par une base de données relationnelle dont le type est a priori indéterminé. De plus, même si notre service informatique décide d'adopter un type de base de données particulier, rien ne garantit la pérennité à moyen et long terme de ce choix.

  L'adoption de bases de données orientées objets n'est pas envisagée.
- **Programmation orientée objet.** La culture en place et les compétences disponibles dans notre service informatique imposent de développer les applications au moyen d'un langage orienté objet. Naturellement, le langage de la base de données doit être accessible au langage de développement.

b) Environnement d'exploitation

- **Applications multi-couches avec client semi-léger.** La logique applicative ("business logic") est distribuée entre le client et le serveur. Le client est donc pourvu d'un minimum d'intelligence.
- **Applications multi-utilisateurs.** Les postes de travail sont nombreux et distants. Une attention particulière doit donc être portée à la gestion de la concurrence: il est impératif que tout utilisateur voulant modifier ou supprimer une donnée de la base de données soit en possession de la dernière version de cette donnée.

- **Types de traitements.** Il s'agit d'applications centrées sur la gestion de données persistentes ("database applications"). Les traitements se limitent donc principalement à la lecture et modification de données.

c) Environnement organisationnel

- **Projet pilote.** En cas de succès, l'architecture choisie sera employée pour le développement d'autres applications. Il est donc fortement suggéré de bien penser son caractère adaptatif.
- **Compétences des parties impliquées dans la réalisation.** Il est indispensable que les choix opérés ne débouchent pas sur une diversification trop importante des technologies. Ceci induirait un risque supplémentaire de taille pour les nouveaux projets: le manque de maîtrise des technologies. Par exemple, l'incertitude liée au choix du type de base de données nous prévient que l'entreprise ne dispose probablement pas de compétences particulières pour un type de base de données spécifique. A terme, l'objectif est de disposer d'un nombre limité de spécialistes et ainsi dispenser les développeurs de devoir maîtriser tous les types de langages offerts par les technologies retenues.

I.2 Délimitation de la problématique

De manière générale, avant d'entreprendre une nouvelle réalisation, il convient de prendre en considération les solutions offertes par le marché, et de mesurer leur adéquation au contexte général choisi et aux besoins préalablement exprimés. Toutefois, dans le cadre de ce travail, nous avons choisi de limiter l'investigation à deux solutions. La première, disponible sur le marché, est constituée de Java et JDBC/SQL. La deuxième est une solution réalisée par nos soins, ARROW.

Notons que la plupart des critiques développées au cours de ce travail concernant la cohabitation Java/JDBC peuvent s'appliquer de manière générale aux situations où un langage de développement orienté objet intègre un module d'accès à des bases de données.

a) Java et JDBC/SQL

JDBC constitue le sujet d'étude idéal. En effet, celui-ci illustre parfaitement la cohabitation entre nos deux paradigmes favoris: le paradigme objet car il fait partie intégrante du langage Java, et le paradigme relationnel car il est avant tout destiné à la manipulation des bases de données relationnelles au moyen du langage SQL. Bien que JDBC soit suffisamment générique que pour permettre un interfaçage avec d'autres types de bases de données (par exemple des fichiers plats), nous allons nous borner à l'évaluer sous son aspect "langage d'accès aux bases de données relationnelles".

b) ARROW

Au lieu de nous "contenter" d'une solution offerte par le marché, nous allons étudier comment un modèle développé sur mesure (ARROW, Architecture for Remote Relational databases Object-oriented Wrapper) peut répondre à nos besoins et aux manquements de JDBC. Il s'agit d'une solution générique qui n'est liée ni à JDBC, ni à Java, même si l'implémentation et les exemples que nous proposerons intègrent ces deux composants. Il est donc important de percevoir que ARROW est un modèle et non une technologie.

c) Autres solutions envisageables

Il est intéressant de remarquer que nous aurions pu choisir d'autres solutions comme support technologique des différents concepts développés dans ce travail. Par exemple, le couple - également assez populaire - C++/ODBC aurait pu convenir. Toutefois, l'objectif de ce travail n'est pas de présenter un éventail complet des solutions existantes auxquels notre contexte général pourrait s'appliquer. Nous nous limiterons donc à la paire Java/JDBC.

I.3 Critères d'évaluation

Afin de systématiser l'étude de chacune des solutions envisagées, nous allons définir un ensemble de critères d'évaluation. Ceux-ci s'inscrivent soit explicitement soit implicitement dans la logique du contexte général et des contraintes d'environnement que nous venons de présenter.

I.3.1 Structure des langages

a) Diversité des langages

Le langage de développement et le langage d'accès aux données sont-ils de la même famille, c'est-à-dire respectent-ils les règles syntaxiques et grammaticales issues du même paradigme? Comment les données sont-elles représentées dans les deux langages? A-t-on un "impedance mismatch" [Heinckiens98], c'est-à-dire une distance sémantique entre les types de données exprimés dans le langage de développement et ceux de la base de données?

b) Complétude des langages

Les fonctionnalités offertes pour la manipulation des données sont-elles suffisantes? Dans quelle mesure peut-on les étendre, et à quel coût? Le langage d'accès aux données permet-il également de manipuler les métas-données? Est-il possible de les consulter et de les modifier (rajouter des tables, des champs, ...)? L'accès aux procédures stockées dans la base de données est-il autorisé?

c) Risques de bogues

Comment l'environnement de développement gère-t-il la cohabitation des deux langages (le langage de développement et celui d'accès aux données)? Le compilateur est-il suffisamment intelligent pour détecter les erreurs formulées dans le langage d'accès aux données? Qu'en est-il de la lisibilité du code?

d) Complexité des langages

Le langage d'accès aux données est-il complexe? Quels sont les pré-requis structurels nécessaires à la manipulation des données? Est-il impératif de construire des structures supplémentaires plus ou moins complexes dans le langage de développement avant de pouvoir accéder aux données?

I.3.2 Fonctionnalités particulières des langages

a) Transactions

La gestion des transactions est-elle destinée uniquement aux modifications appliquées à la base de données ou bien est-elle intégrée au langage de développement, de telle sorte que les variables et objets déclarés puissent également y participer?
Est-il possible de démarrer plusieurs transactions en parallèle? Les transactions imbriquées sont-elles autorisées?

b) Événements

Existe-t-il un mécanisme permettant de capter des événements en provenance de la base de données, par exemple suite à la manipulation par un autre client de l'enregistrement sur lequel on est connecté?

c) Résolution des conflits

Quelles sont les techniques proposées par le langage d'accès aux données pour gérer les situations conflictuelles lors des mises à jour? Lorsque plusieurs utilisateurs veulent modifier la même donnée simultanément, comment la priorité est-elle accordée? Y a-t-il une possibilité d'empêcher préventivement d'autres utilisateurs de modifier une donnée?

d) Portabilité

Le langage d'accès aux données est-il portable vers d'autres types de bases de données? Le changement de type de base de données implique-t-il la réécriture d'une partie ou de la totalité de l'application? Impacte-t-il uniquement l'aspect configuration de l'application, donc sans qu'il soit nécessaire de modifier une seule ligne de code?

e) Performance

Les fonctionnalités offertes sont-elles performantes? Est-on en présence d'une surcharge causée par l'utilisation du langage d'accès aux données à partir du langage de développement?

f) Répartition

Quelles sont les contraintes d'architecture liées à la réalisation d'une application multi-couches et répartie où les clients sont semi-légers? La répartition est-elle transparente pour les langages de développement et d'accès aux données?

I.4 Base de données illustrative

Les exemples développés au cours de ce travail s'appuient sur une base de données relationnelle illustrative. La Figure I-1 présente son schéma conceptuel sous la forme d'un diagramme de classes.

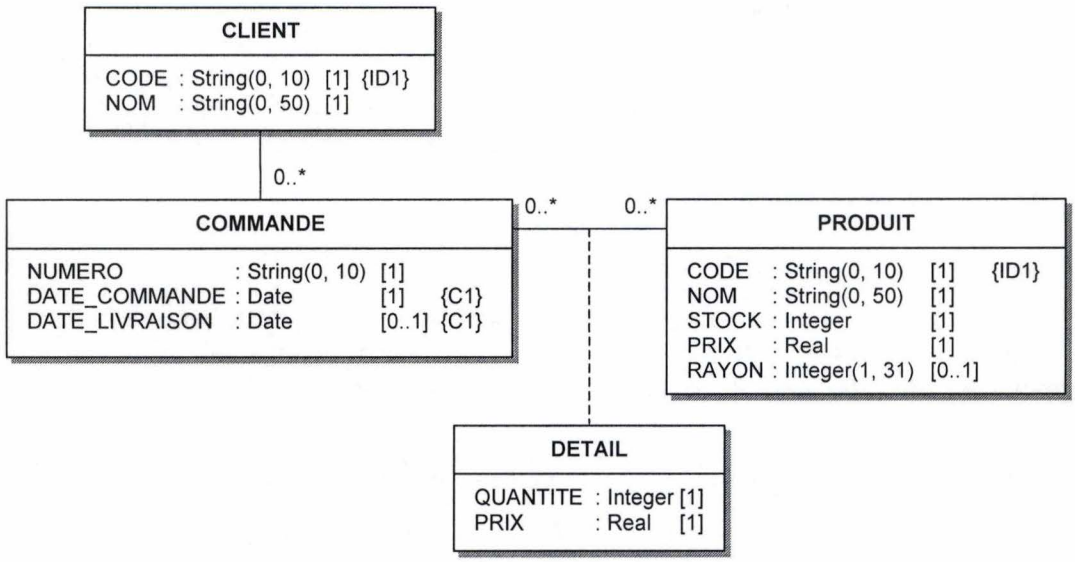

Figure I-1 Schéma conceptuel de la base de données illustrative

La Figure I-2 illustre le schéma physique. Celui-ci se caractérise par l'apparition d'identifiants techniques (clés primaires et étrangères) et de champs dédiés à la gestion de la concurrence (numéros de versions des enregistrements).

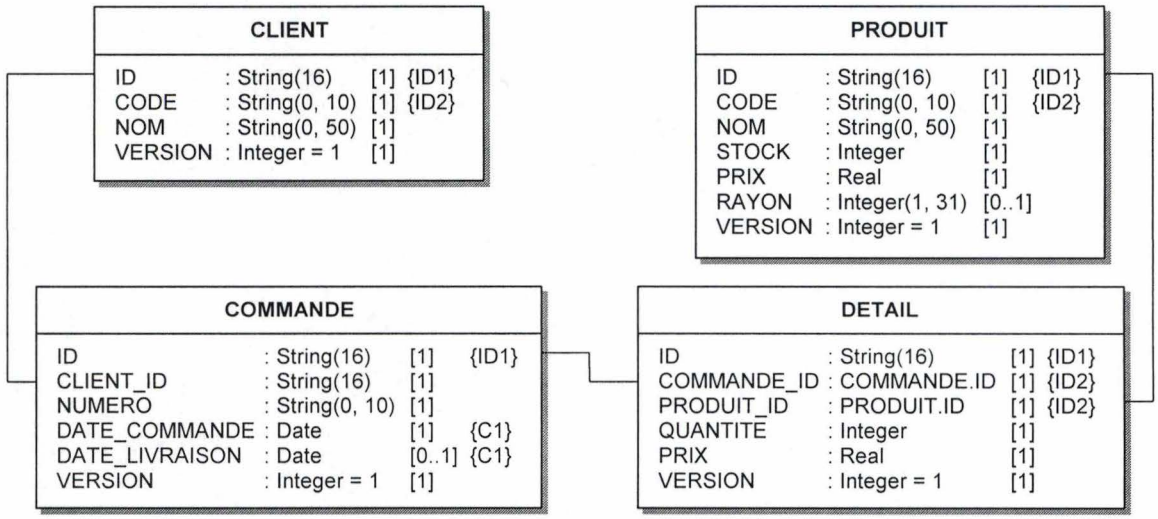

Figure I-2 Schéma physique de la base de données illustrative

On notera en particulier que:

- La contrainte {IDx} définit les champs participant à la composition de l'identifiant x. Techniquement, les {ID1} représentent ici les clés primaires, alors que les {ID2..*} représentent des clés secondaires.
- Le champ *VERSION*, présent dans toutes les tables, contient une valeur indiquant le nombre de mises à jours effectuées sur chacun des enregistrements. Sa valeur initiale de 1 est incrémentée d'une unité lors de chaque mise à jour. Son rôle est de gérer les accès concurrents aux enregistrements (cfr II.3.2.c).
- Le domaine de valeurs des champs de type *String* (chaîne de caractères) est exprimé sous la forme *String(longueur minimum, longueur maximum)* ou *String(longueur imposée)*.
- Le domaine de valeurs du champ *PRODUIT.RAYON* est limité à l'intervalle [1..31]
- La contrainte {C1} exprime que le champ *COMMANDE.DATE_LIVRAISON*, s'il est garni, doit être supérieur ou égal à *COMMANDE.DATE_COMMANDE*.
- Les noms de tables et de champs sont exprimés en majuscules afin de faciliter la distinction avec les éléments de l'architecture que nous allons décrire au cours de ce travail, lesquels respectent les règles syntaxiques de Java.

II
Le modèle JDBC

Dans ce chapitre, nous présentons succinctement l'architecture et le fonctionnement du modèle JDBC. Nous procédons ensuite à son évaluation sur la base des critères définis dans le chapitre précédent. Les critiques formulées nous serviront de support pour la définition des spécifications du modèle alternatif (ARROW) que nous proposerons dans le chapitre suivant.
Pour une description détaillée du modèle JDBC, le lecteur s'en remettra à la documentation technique fournie avec le SDK (Software Developer Kit) de Java [JavaSDK13] ou à des ouvrages traitant le sujet [JavaProg00 et Weber99].

II.1 Architecture

II.1.1 Configuration logique

Toute application ou composante d'application, qui requiert les services de JDBC, est décomposable en trois couches logiques: un ou plusieurs clients, un ou plusieurs pilotes JDBC et une ou plusieurs bases de données, comme l'illustre la Figure II-1.

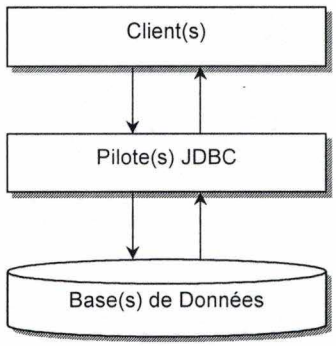

Figure II-1: Les trois couches d'une application architecturée autour de JDBC

Notons que tout pilote JDBC est constitué de deux sous-couches. La première, que l'on appelle "JDBC API (Application Program Interface)", est l'interface entre le client Java et le pilote. Cette interface est identique quel que soit le type de la base de données sous-jacente. La deuxième sous-couche est spécifique au type de la base de données et joue le rôle d'interprète entre la première couche et la base de données.

II.1.2 Configuration physique

Outre la division logique de la Figure II-1, plusieurs configurations physiques sont envisageables. Nous en présentons deux. La première consiste à intégrer le client et le pilote JDBC dans le même environnement d'exécution. Dans ce cas, tous les traitements du client et du pilote sont exécutés dans la même machine virtuelle. L'exécution des requêtes proprement dites reste du ressort de la base de données - qui est généralement distante - sauf par exemple lorsque le pilote JDBC est utilisé pour manipuler des fichiers plats.
La deuxième solution est d'utiliser un type particulier de pilote JDBC: un serveur de pilotes ("middleware"). Celui-ci peut naturellement se trouver dans un environnement d'exécution distant. Dans ce cas, le pilote du client se contente d'envoyer les requêtes au serveur qui se charge alors du traitement et de la communication avec la base de données. Ces deux scenarios sont illustrés par la Figure II-2.

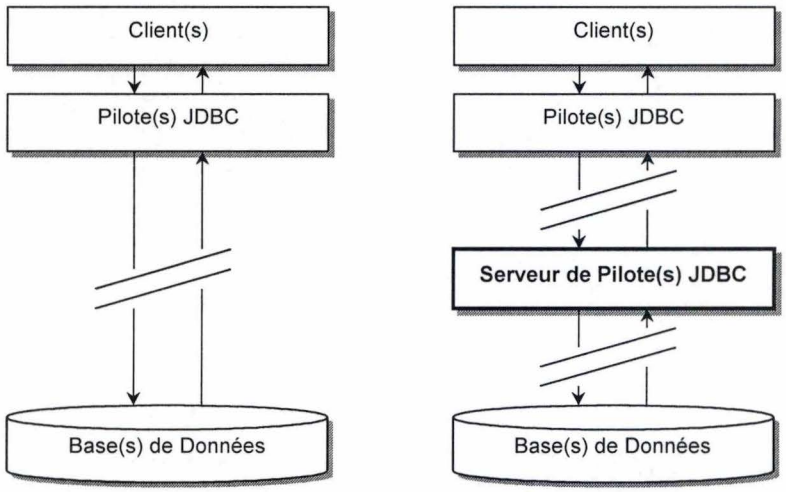

Figure II-2: Répartition des composants d'une application architecturée autour de JDBC à travers plusieurs environnements d'exécution habituellement (mais pas obligatoirement) distants.

Il existe bien sûr d'autres alternatives, mais qui requièrent l'intervention de composants tiers, externes au cadre que nous avons délimité. Nous ne les envisagerons donc pas.

Nous allons maintenant examiner comment le client communique avec un pilote JDBC pour transmettre ses requêtes à la base de données et réceptionner les résultats. La connexion entre un pilote et la base de données sous-jacente et/ou un serveur de pilote ne sera plus évoquée.

II.2 Fonctionnement

Nous allons d'abord présenter les grandes lignes du protocole permettant à un client d'exploiter les services offerts par JDBC, pour ensuite l'illustrer par un exemple pratique.

II.2.1 Description du protocole

Le langage de JDBC met à la disposition du développeur un ensemble de fonctionnalités sous la forme de classes et de méthodes, certaines d'entre elles acceptant des requêtes SQL en paramètres. Lorsque le client désire transmettre une requête à une base de données via un pilote JDBC, il est obligatoire de suivre une procédure prédéfinie (un protocole), constituée d'un certain nombre d'étapes dépendant du type de la requête. Chacune de ces étapes est illustrée par la Figure II-3.

- Tout d'abord, le client charge le pilote JDBC désiré (0).
- Le client demande alors au pilote JDBC d'établir une connexion vers la base de données sous-jacente (1). En cas de réussite, le pilote retourne au client un objet représentant la connexion ouverte vers la base de données (2).
- Ensuite, le client demande au pilote de lui fournir une structure d'accueil - un objet "statement" - (3) nécessaire à la transmission et à l'exécution ultérieure de la requête. Sur ce, le pilote crée un objet "statement" et le retourne au client (4).

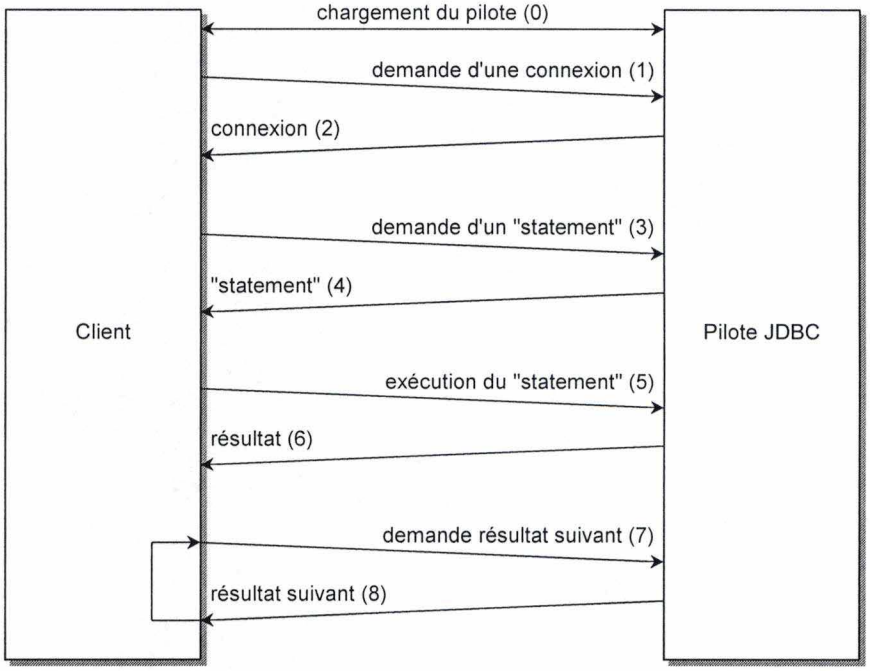

Figure II-3: Interaction schématisée entre un client et un pilote JDBC

- Le client garnit alors le "statement" avec la requête souhaitée (par exemple une requête SQL ou l'appel à une procédure stockée) et demande au pilote de l'exécuter (5). Le pilote réceptionne la requête, s'assure qu'elle est syntaxiquement correcte et effectue d'éventuelles optimisations. Ensuite, certains éléments de la requête sont convertis dans un langage spécifique à la base de données sous-jacente. La requête est envoyée à la base de données qui l'exécute et retourne le résultat au pilote (5). Le pilote à son tour renvoie les résultats au client. La nature des résultats est fonction du type de requête. Pour les requêtes de mise à jour, le pilote retourne le nombre d'enregistrements manipulés (c'est-à-dire modifiés, créés ou supprimés) et pour les requêtes de consultation, un objet représentant les enregistrements sélectionnés (6).
- Si la requête est de type consultation, le client peut parcourir les enregistrements sélectionnés en demandant au pilote de fournir le résultat suivant. La première demande de résultat suivant retourne le premier enregistrement. Le pilote retourne donc le résultat (en fait, l'enregistrement) si la fin de la liste des enregistrements sélectionnés n'a pas été dépassée, sinon il ne retourne rien (7 et 8). Les demandes ultérieures retournent les enregistrements suivants.

Toute erreur détectée lors d'un échange est retournée au client sous la forme d'une exception, par exemple lorsque la base de données n'est plus accessible, si la requête formulée est invalide, ...

II.2.2 Exemple pratique

La Figure II-4 illustre les échanges protocolaires entre un client et un pilote JDBC dans le cadre d'une requête comptant le nombre d'enregistrements de la table *CLIENT* de notre base de données illustrative.

```
try
{
  // Etape 0
  // Le client charge un pilote JDBC
  Class.forName("<pilote JDBC>");

  // Étapes 1 & 2
  // La connexion avec la base de données est établie
  Connection con = DriverManager.getConnection("<base de données>");

  // Etapes 3 & 4
  // Un objet de type Statement est fourni par le pilote
  Statement sta = con.createStatement();

  // Etapes 5 & 6
  // Une requête est envoyée au pilote et le résultat est retourné
  ResultSet res = sta.executeQuery("SELECT COUNT(*) FROM CLIENT");

  // Etapes 6 & 7
  // On demande au pilote de fournir le résultat suivant (en fait
  // le premier résultat)
  res.next();

  // On exploite le résultat
  System.out.println(res.getInt(1));
}
catch(Exception e)
{
  // Réaction du programme en cas d'erreur
  e.printStackTrace();
}
```

Figure II-4 Illustration de l'interaction entre un client et un pilote JDBC

Pour envoyer une nouvelle requête à la même base de données via le même pilote JDBC, il n'est généralement pas nécessaire de réitérer les étapes 0 à 4, sauf dans certaines situations où par exemple l'on désire démarrer une nouvelle transaction en parallèle ou si l'on travaille avec des requêtes paramétrées (cfr plus loin, Figure II-12).

II.3 Évaluation du modèle

Nous allons procéder à l'évaluation du modèle JDBC sur la base des critères que nous avons définis plus haut (cfr I.3). Certains d'entre eux constituent un manquement plus ou moins sévère auquel nous nous efforcerons de répondre, au moyen du modèle alternatif que nous proposerons.

Chaque critère évalué fait l'objet d'une pondération suivant une échelle de trois valeurs définies comme suit:

2/2 : L'évaluation du critère donne un résultat plus que satisfaisant
1/2 : L'évaluation du critère donne un résultat satisfaisant
0/2 : L'évaluation du critère donne un résultat insatisfaisant

Ces valeurs sont indiquées entre parenthèses derrière les noms des critères évalués. Seul le résultat 2/2 certifie l'adéquation de JDBC au cadre fixé (cfr I.1). Les deux autres pondérations témoignent de lacunes.

II.3.1 Structure des langages

a) Diversité des langages (1/2)

Sous JDBC, le développeur est confronté à deux paradigmes différents: objet (Java) et relationnel (SQL). En réalité, JDBC en tant que modèle est effectivement orienté objet, mais les données qu'il manipule à destination ou en provenance de la base de données ne le sont pas.
Cette cohabitation alourdit sans aucun doute le processus de développement. Elle impose au développeur de composer non seulement avec deux langages différents (qu'il se doit de maîtriser) , mais également avec deux modes de représentation des données ("impedance mismatch" [Heinckiens98]). Il doit s'intéresser aux conversions des types de données entre l'environnement de développement et la base de données, comme l'illustre la Figure II-5.

```
try
{
  Class.forName("<pilote JDBC>");

  Connection con = DriverManager.getConnection("<base de données>");
  Statement  sta = con.createStatement();
  ResultSet  res = sta.executeQuery("SELECT COUNT(*) FROM CLIENT");

  res.next();
  System.out.println(res.getByte(1));
}
catch(Exception e)
{
  e.printStackTrace();
}
```

Figure II-5 Exemple où le type de donnée "byte" sera certainement insuffisant. Le type de donnée retourné par la requête de comptage n'étant pas explicite, c'est au développeur de choisir celui qui convient.

Notons également que les outils de génération de documentation automatique comme JavaDoc sont dans ce contexte incapables de réconcilier la structure de données de l'application cliente avec celle de la base de données. Seule une documentation annexe ou un parcours du code permet d'établir le lien entre les deux environnements.

b) Complétude des langages (2/2)

Dans l'absolu, l'analyse de la complétude d'un langage est ardue voire impossible à réaliser. Elle relève de critères dépendant du contexte d'utilisation et est donc subjective. Nous allons dès lors postuler que le couple Java/SQL implémenté par JDBC est à même de remplir les fonctionnalités attendues dans le cadre que nous nous sommes fixé.
Ainsi, JDBC permet à un client de communiquer avec une base de données via le langage SQL offert par celle-ci, de parcourir la structure d'une base de données (méta-données) et d'exécuter des procédures stockées.

c) Risques de bogues (0/2)

Puisque les requêtes SQL sont formulées sous la forme de chaînes de caractères, le compilateur Java est totalement incapable d'en contrôler la syntaxe. Il s'agit là d'une source de bogues potentiels car les éventuelles erreurs ne seront détectées qu'à l'exécution. La Figure II-6 illustre cet aspect.

```
try
{
  Class.forName("<pilote JDBC>");

  Connection con = DriverManager.getConnection("<base de données>");
  Statement  sta = con.createStatement();
  ResultSet  res = sta.executeQuery("SELECT CONT(*) FROM CLIENT");
  res.next();

  System.out.println(res.getString(1));
}
catch(Exception e)
{
  e.printStackTrace();
}
```

Figure II-6 Exemple de code contenant deux erreurs indécelables par le compilateur. La première est une erreur de syntaxe dans la formulation de la requête SQL (*CONT* doit être changé en *COUNT*); la deuxième est une erreur de sémantique car la valeur à récupérer du *ResultSet* est un entier et non une chaîne de caractères (*getString(1)* doit être remplacé par *getInt(1)*).

Notons également que les outils de développement proposant des fonctionnalités du type "intellisense" (auto-complétion et auto-vérification du code) restent impuissants face à ce type d'erreur.

La cohabitation Java/SQL pose également des problèmes de lisibilité et est par là une source de bogues supplémentaire. La Figure II-7 illustre comment le passage de paramètres est possible entre le langage Java et SQL dans le cas d'une requête de création.

```
try
{
  Class.forName("<pilote JDBC>");

  Connection con = DriverManager.getConnection("<base de données>");
  PreparedStatement sta = con.prepareStatement(
    "INSERT INTO CLIENT(ID, CODE, NOM, VERSION) VALUES(?, ?, ?, ?)");

  sta.setString(1, "12345678");
  sta.setString(2, "TOUR");
  sta.setString(3, "TOURNESOL TRYPHON");
  sta.setInt(4, 1);

  System.out.println("Nombre de créations: " + sta.executeUpdate());
}
catch(Exception e)
{
```

```
  e.printStackTrace();
}
```

Figure II-7 Exemple de mauvaise lisibilité du code. Il faut parcourir la requête SQL pour déterminer le numéro d'ordre de chacun des champs, pour pouvoir ensuite leur assigner une valeur au moyen des méthodes *setString*(...) et *setInt*(...) de l'objet *Statement*.

Même si la lisibilité est un critère plutôt subjectif et surtout proportionnel à la qualité de la mise en forme du programme, elle est essentielle à la production d'applications de qualité car elle en facilite la maintenance.

d) Complexité des langages (1/2)

Chaque clause SQL possède sa propre grammaire. Par exemple, une requête de création est sensiblement différente d'une requête de modification, comme l'illustre la Figure II-8.

```
INSERT
    INTO CLIENT(ID, CODE, NOM, VERSION)
    VALUES("12345678", "TOUR", "TOURNESOL SERAPHIN", 1);

UPDATE
    CLIENT
    SET CODE    = "TOUR",
        NOM     = "TOURNESOL TRYPHON",
        VERSION = VERSION + 1
    WHERE ID = "12345678";
```

Figure II-8 Exemple de diversité dans dans la grammaire des clauses SQL

Même si le but de ces deux requêtes est de mettre un enregistrement à jour dans la base de données (la première crée un enregistrement, la deuxième le modifie), leur formulation est complètement différente. En fait, on pourrait imaginer une grammaire commune pour les clauses INSERT et UPDATE, comme illustré par la clause INSERT fictive dans l'exemple de la Figure II-9.

```
INSERT INTO
    CLIENT
    SET ID      = "12345678",
        CODE    = "TOUR",              /* !!! Clause INSERT fictive !!! */
        NOM     = "TOURNESOL SERAPHIN",
        VERSION = 1;

UPDATE
    CLIENT
    SET CODE    = "TOUR",
        NOM     = "TOURNESOL TRYPHON",
        VERSION = VERSION + 1
    WHERE ID = "12345678";
```

Figure II-9 Exemple de requête INSERT fictive utilisant une grammaire analogue à la clause UPDATE

Sous JDBC, on est confronté non seulement à la diversité des grammaires des clauses SQL (comme nous venons de le voir), mais aussi au fait que Java impose au développeur d'utiliser des structures distinctes pour la composition des requêtes et pour la gestion des résultats, comme illustré par l'exemple de la Figure II-10.

```
try
{
  Class.forName("<pilote JDBC>");

  Connection con = DriverManager.getConnection("<base de données>");

  // Exemple de création
  PreparedStatement sta = con.prepareStatement(
    "INSERT INTO CLIENT(ID, CODE, NOM, VERSION) VALUES(?, ?, ?, ?)");

  sta.setString(1, "12345678");
  sta.setString(2, "CAST");
  sta.setString(3, "CASTAFIORE BIANCA");
  sta.setInt(4, 1);

  System.out.println("Nombre de créations: " + sta.executeUpdate());

  // Exemple de consultation
  Statement sta = con.createStatement();
  ResultSet res = sta.executeQuery(
    "SELECT ID, CODE, NOM, VERSION " +
    "FROM CLIENT " +
    "WHERE ID = '12345678'");

  res.next();

  System.out.println("Id      : " + res.getString("ID"));
  System.out.println("Code    : " + res.getString("CODE"));
  System.out.println("Nom     : " + res.getString("NOM"));
  System.out.println("Version : " + res.getInt("VERSION"));
}
catch(Exception e)
{
  e.printStackTrace();
}
```

Figure II-10 Illustration de la diversité des structures Java dédiées à la manipulation des données (ici un objet *Statement* et un objet *ResultSet*), alors que celles-ci sont de même nature, puisque d'un point de vue sémantique, il s'agit de données à destination ou en provenance de la table *CLIENT*.

Admettons toutefois que la préparation des structures nécessaires à l'exécution d'une requête reste raisonnable, sachant que, comme nous l'avons évoqué dans la description du protocole JDBC (cfr II.2), certaines étapes comme le chargement du pilote JDBC et l'établissement de la connexion avec la base de données ne sont pas toujours indispensables (sauf bien sûr lors de la première exécution).

La complexité des langages, induite par la diversité de leurs grammaires, est généralement techniquement et/ou fonctionnellement justifiable. Nous les retenons comme défaut mineur du modèle JDBC car la formation des développeurs peut naturellement pallier ces difficultés.

II.3.2 Fonctionnalités particulières des langages

a) Transactions (1/2)

La gestion des transactions est réservée aux données de la base de données. L'annulation d'une transaction n'a donc aucune influence sur les variables locales. Les requêtes imbriquées ne sont pas supportées. Il est par contre possible de démarrer plusieurs transactions en parallèle à la condition d'ouvrir autant de connexions vers la base de données que de transactions.

b) Événements (0/2)

JDBC ne gère pas l'envoi de messages de notification ("triggers"), par exemple suite à une mise à jour ou une suppression d'enregistrement.

c) Résolution des conflits (1/2)

La gestion des conflits peut se scinder en deux catégories: la gestion a priori et la gestion a posteriori.

La gestion a priori consiste en un blocage préventif des enregistrements à manipuler (modifier ou supprimer). Le blocage est en général assorti d'une lecture des enregistrements, comme c'est le cas en SQL avec les requêtes "*SELECT ... FOR UPDATE*". On est ainsi assuré de disposer de la dernière version des enregistrements. Ces requêtes ne sont malheureusement pas supportées par tous les types de pilotes JDBC.

La gestion a posteriori consiste à vérifier, lors de la mise à jour, que l'enregistrement n'a pas été modifié par un autre client. Cette solution requiert la présence d'un champ de type "timestamp" dont la valeur change à chaque modification de l'enregistrement. Ce champ est tel que deux modifications (quelle que soit leur dispersion dans le temps) sur le même enregistrement ne peuvent générer la même valeur.
Lorsqu'un client désire modifier un enregistrement, il vérifie simultanément (cette manipulation doit être atomique, par exemple via une requête "*UPDATE ... WHERE ...*", voire une *requête "SELECT ... WHERE ... FOR UPDATE*" suivie d'une requête "*UPDATE...*") que la valeur de ce champ dans la base de données est bien identique à la valeur qu'il avait préalablement chargée, par exemple pour afficher les données à l'écran.

Dans les deux cas, les conflits sont gérés explicitement en SQL et font donc l'objet d'une attention particulière du développeur. Une gestion implicite serait sans aucun doute plus confortable.

d) Portabilité (0/2)

L'utilisation de SQL propriétaire à la base de données manipulée risque de rendre l'application difficilement portable vers d'autres bases de données. L'exemple de la Figure II-11 illustre cet aspect.

```
try
{
  Class.forName("<pilote JDBC>");

  Connection con = DriverManager.getConnection("<base de données>");
  Statement  sta = con.createStatement();
  ResultSet  res = sta.executeQuery(
    "SELECT DISTINCT TO_CHAR(DATE_COMMANDE, 'MON-DD-YYYY')
     FROM COMMANDE");

  while(res.next())
    System.out.println(res.getString(1));
}
catch(Exception e)
{
  e.printStackTrace();
}
```

Figure II-11 Exemple de requête SQL spécifique à un type de base de données: la clause *TO_CHAR* est spécifique à Oracle et est donc incompatible avec MySql ou SqlServer.

Pour assurer la portabilité, le développeur est alors contraint d'utiliser une syntaxe SQL commune à toute base de données, ou bien de paramétrer la construction des requêtes SQL en fonction du type de base de données à interroger.

La résolution a priori des conflits (cfr ci-dessus) ainsi que la répartition (cfr plus loin) apportent d'autres éléments au désavantage de la portabilité.

e) Performance (2/2)

Nous considérons ici que le modèle JDBC offre une technologie suffisamment performante pour les besoins que nous avons définis (cfr I.1). Dans d'autres contextes d'implantation, il pourrait se révéler totalement sous-performant.

Toutefois, l'exploitation efficiente de JDBC est tributaire du niveau de compétence du développeur. Il est par exemple absurde d'ouvrir une nouvelle connexion vers la base de données à chaque transmission de requête. Nous allons soulever ici un autre aspect relatif à la performance: la pré-validation.

Lors d'une requête de mise à jour, le contrôle d'intégrité est réalisé par la base de données elle-même alors que certaines vérifications (par exemple: respect des domaines de valeurs, de la cardinalité, des règles d'exclusion, ...) pourraient être effectuées préalablement par le client. La Figure II-12 ci-dessous nous propose un exemple.

```
public int createClient(String sId, String sCode, String sNom)
{
  try
  {
    Class.forName("<pilote JDBC>");

    Connection con = DriverManager.getConnection("<base de données>");
    PreparedStatement sta = con.prepareStatement(
      "INSERT INTO CLIENT(ID, CODE, NOM, VERSION) VALUES(?, ?, ?, ?)");

    sta.setString(1, sId);
    sta.setString(2, sCode);
    sta.setString(3, sNom);
    sta.setInt(4, 1);

    return(sta.executeUpdate());
  }
  catch(Exception e)
  {
    e.printStackTrace();
    return(0);
  }
}
```

Figure II-12 Exemple d'inefficience se traduisant par une surcharge inutile du réseau et de la base de données. Si par exemple la valeur de *sNom* est nulle, la requête est quand-même envoyée à la base de données qui refusera la mise à jour et déclenchera une exception. De plus, l'application cliente est mise en attente alors que la requête sera de toute façon refusée.

Afin de soulager la tâche de la base de données, d'économiser les ressources réseau et d'optimiser la réactivité de l'application cliente (n'oublions pas que nous travaillons dans un environnement où les clients sont distants), il faudrait éviter de transmettre des requêtes invalides pour lesquelles une pré-validation est envisageable sur le client. Toutefois, l'objet ici n'est pas de détacher les règles d'intégrité de la base de données, mais de profiter de l'intelligence du client pour effectuer une vérification préalable au contrôle d'intégrité exécuté par la base de données.

Une optimisation rapide de l'exemple précédent consisterait à rajouter des lignes de code dédiées à la vérification de la validité de chacun des paramètres, comme proposé par la Figure II-13.

```
public int createClient(String sId, String sCode, String sNom)
{
  try
  {
    checkClient(sId, sCode, sNom);

    ... cfr Figure II-12 ...
  }
  catch(Exception e)
  {
    e.printStackTrace();
    return(0);
  }
}

public void checkClient(String sId, String sCode, String sNom)
{
  if (sId == null || sCode == null || sNom == null)
    throw(new NullPointerException());
}
```

Figure II-13 Exemple d'optimisation élémentaire. Si un des paramètres est nul, une exception est immédiatement déclenchée. Aucun accès à la base de données n'est alors nécessaire.

Ce type d'optimisation est applicable dans un contexte réel à la condition sine qua non que le contrôle de validité soit centralisé. Le non-respect de cette règle aboutirait à des incohérences et à des problèmes de maintenance. Dans l'exemple de la Figure II-13, la méthode *checkClient* se charge de centraliser le contrôle de validité.

Toutefois, même si cette démarche peut paraître satisfaisante, elle impose la présence de méthodes de création, de suppression, ... pour chaque table. Ainsi, pour permettre la mise à jour des enregistrements de la table *CLIENT*, il faudrait au moins définir les méthodes *checkClient*, *createClient*, *updateClient*, *deleteClient* et procéder de la même manière pour chacune des tables. De plus, puisque les valeurs de champs sont passées en paramètre, on peut se retrouver face à des méthodes en comportant plusieurs dizaines, ce qui ne participe pas à la lisibilité du code.

f) Répartition (0/2)

Nous avons vu plus haut (cfr II.1) que JDBC offre la possibilité de travailler avec un serveur de pilotes assurant la répartition et ce, de manière quasi-transparente pour le développeur. Malheureusement, tous les types de pilotes n'offrent pas cette fonctionnalité. On est alors contraint d'acquérir ou de développer un "middleware" jouant le rôle d'intermédiaire entre le client et le pilote JDBC, comme présenté sur la Figure II-14.

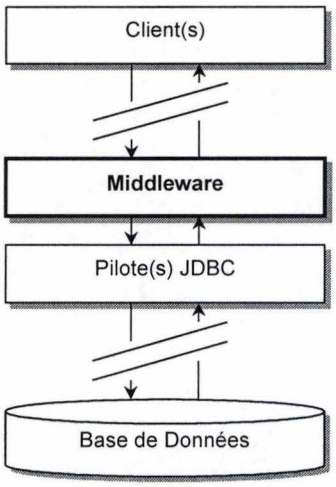

Figure II-14: Topologie typique lorsque le type de pilote JDBC n'offre pas la possibilité de fonctionner en mode serveur.

Le pilote JDBC n'est plus directement accessible au client. Le développeur doit donc composer avec le langage du "middleware". En fait, l'architecture que nous allons proposer s'inspire de ce type de configuration.

II.3.3 Résultat global

La faible pondération de la plupart des critères évalués nous interpelle. Tel quel, le modèle JDBC est difficilement conciliable avec nos objectifs. Dès lors, un double choix s'offre à nous: soit nous écartons définitivement le modèle JDBC et nous recherchons une autre solution, soit nous confinons la puissance de JDBC dans une solution propriétaire conçue par nos soins. Nous allons retenir cette deuxième option et entamer sa réalisation dans le chapitre suivant.

III

Spécifications du modèle ARROW

En réponse aux lacunes du modèle JDBC étudiées dans le chapitre précédent, nous allons définir les spécifications d'un modèle alternatif. Nous commençons par l'étude de son architecture, suivie de la description des fonctionnalités attendues de chacun de ses composants. Nous terminerons par une présentation des limites et extensions potentielles.

III.1 Architecture

Signalons d'emblée que notre objectif n'est pas de modéliser une base de données orientée objet, puisque la persistance reste assurée par une base de données relationnelle. ARROW est un "middleware" constitué d'une couche client et d'un moteur. La nature des services offerts par ces couches fait l'objet de ce chapitre.

III.1.1 Middleware

L'architecture de ARROW dissimule au client quel pilote d'accès aux données est réellement utilisé.

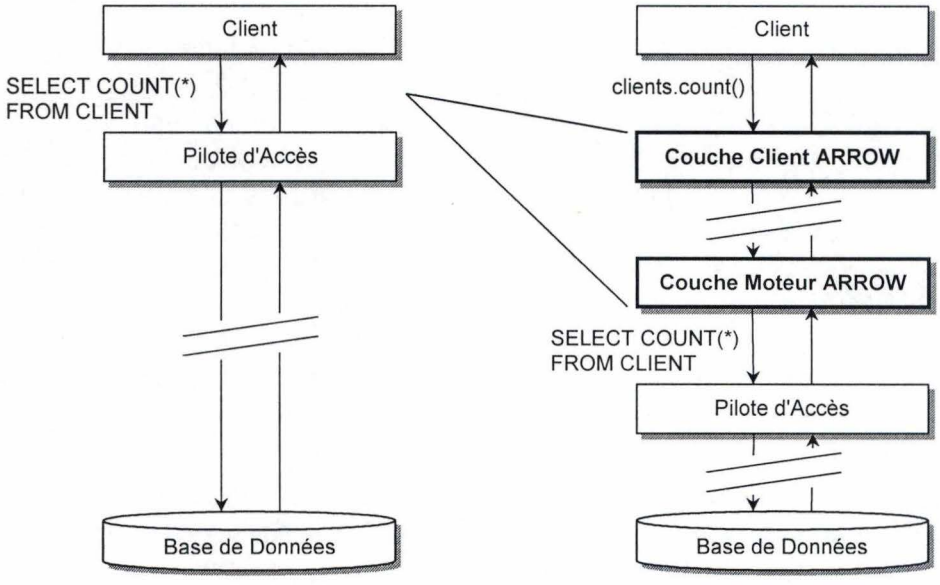

Figure III-1 Exemple de dissimulation du pilote d'accès aux données (par exemple: JDBC) aux yeux du client, en insérant deux couches intermédiaires entre le client et le pilote d'accès.

La Figure III-1 présente à gauche un schéma classique d'architecture basée sur un pilote d'accès aux données tel que JDBC (il pourrait très bien s'agir d'un autre type de pilote), et à droite un schéma intégrant les deux couches de ARROW.

On constate donc que le pilote d'accès n'est pas remplacé; il est dissimulé aux yeux du client. A gauche, le client communique en SQL avec la base de données via le pilote d'accès. A droite le client dialogue avec la base de données par l'intermédiaire de notre architecture à laquelle il transmet ses commandes dans un langage orienté objet, lesquelles seront ensuite traduites en SQL et envoyées au pilote d'accès.

III.1.2 Structure en deux couches

La découpe de ARROW en deux couches vient naturellement du fait qu'il est nécessaire de dialoguer d'une part avec le client et d'autre part avec le pilote d'accès aux données.

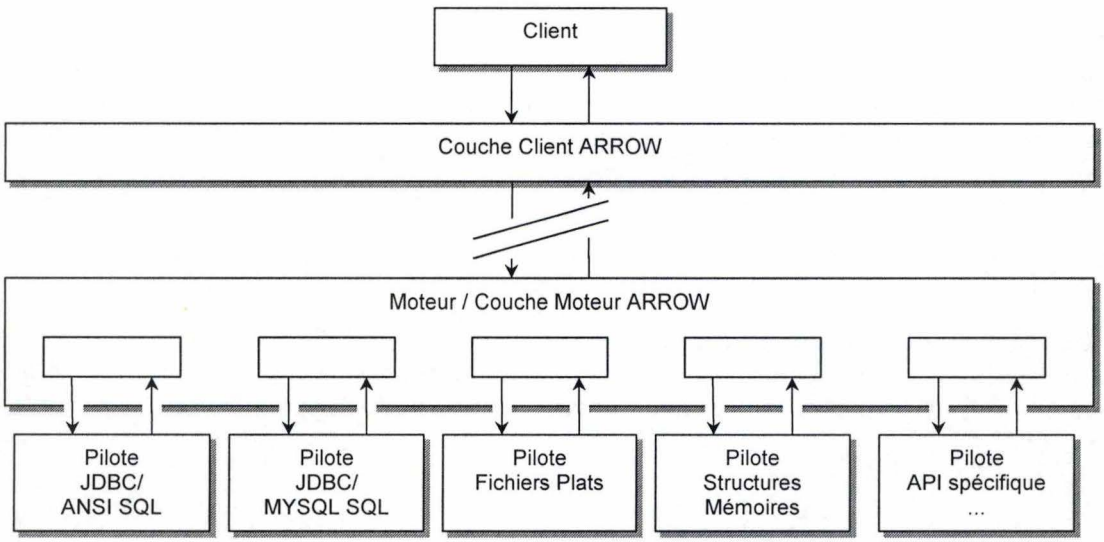

Figure III-2 Structure ARROW en deux couches. La couche client reste identique quel que soit le pilote d'accès aux données utilisé par le moteur.

Approfondissons la description de ces deux couches illustrées par la Figure III-2:

- **Couche client.** Cette couche met un langage à la disposition du client pour lui permettre de transmettre ses ordres à la base de données sous-jacente. Puisque nous nous situons dans un environnement objet, ce langage est composé de classes, d'interfaces, de méthodes et de propriétés. Les ordres en provenance du client sont contrôlés (vérification des contraintes d'intégrité intra-enregistrement, pertinence de l'ordre, ...), avant d'être soit rejetés, soit transmis tels quels au moteur qui assure leur traitement et le retour des résultats (ou des erreurs), lesquels sont alors mis à la disposition du client.

  La couche client est totalement isolée de tout pilote d'accès aux données. Elle peut se localiser dans un environnement d'exécution distinct et distant (nous étudions la répartition plus loin, cfr III.1.3). En d'autres termes, le client ignore totalement si la persistance est gérée par une base de données relationnelle ou un gestionnaire de fichiers plats.

- **Moteur / couche moteur (serveur d'objets).** Cette couche traduit les ordres en provenance de la couche client dans un langage compréhensible par le pilote cible d'accès aux données (gestionnaire de persistance), et les transfère ensuite vers ce pilote d'accès. Les résultats (ou erreurs) d'exécutions, retournés par le pilote cible, sont ensuite renvoyés à la couche client après avoir été retraduits dans son langage.

L'architecture doit permettre au moteur de gérer un nombre indéfini de pilotes d'accès aux données. Le schéma de la Figure III-2 nous en présente quatre: JDBC avec du SQL ANSI, JDBC avec du SQL propriétaire à MYSQL, un gestionnaire de fichiers plats et un gestionnaire de structures mémoires (stockage dans des arbres, vecteurs, ...). Si le premier des exemples (JDBC/ANSI SQL) a fait l'objet d'une implémentation dans le cadre de ce travail, les autres sont donnés à titre informatif. Notons que la complexité du processus de transformation des ordres en provenance de la couche client est évidemment liée au pilote d'accès choisi.

En résumé, on peut dire que, grâce au moteur, l'architecture peut être greffée à n'importe quel type et schéma de base de données. Le langage qu'elle propose au développeur (interface client) reste totalement indépendant de toute implémentation spécifique à un type ou schéma de base de données particulier et n'exige aucune modification structurelle sur la base de données à manipuler (il faut cependant disposer d'un moteur connectable au type de base de données souhaité).

III.1.3 Répartition

Nous venons de voir que notre architecture est composée de deux couches. Celles-ci sont distribuables à volonté comme l'illustre la Figure III-3, où les environnements d'exécution sont encadrés en pointillé. Chaque environnement d'exécution peut naturellement se situer sur une machine distante.

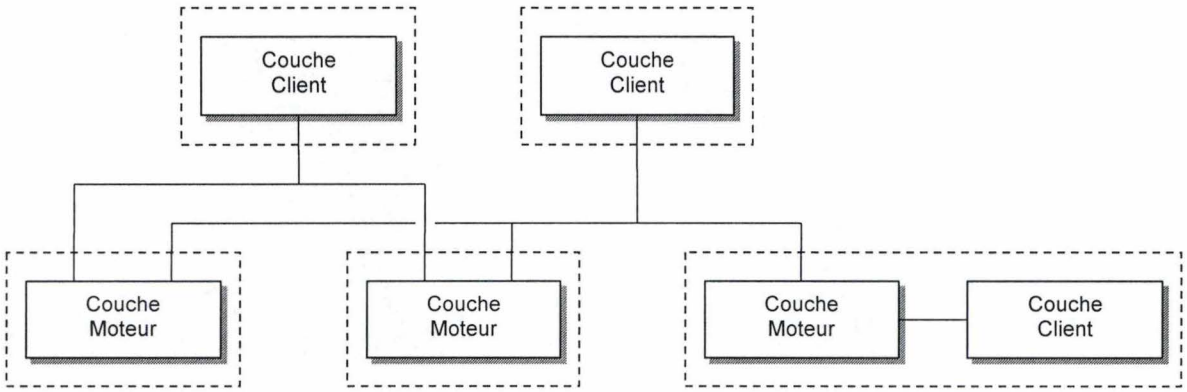

Figure III-3 Exemple de répartition des couches de l'architecture. On constate la diversité des combinaisons offertes.

La base de données même peut être localisée sur un serveur tiers, voire répartie, comme l'illustre l'exemple de la Figure III-4.

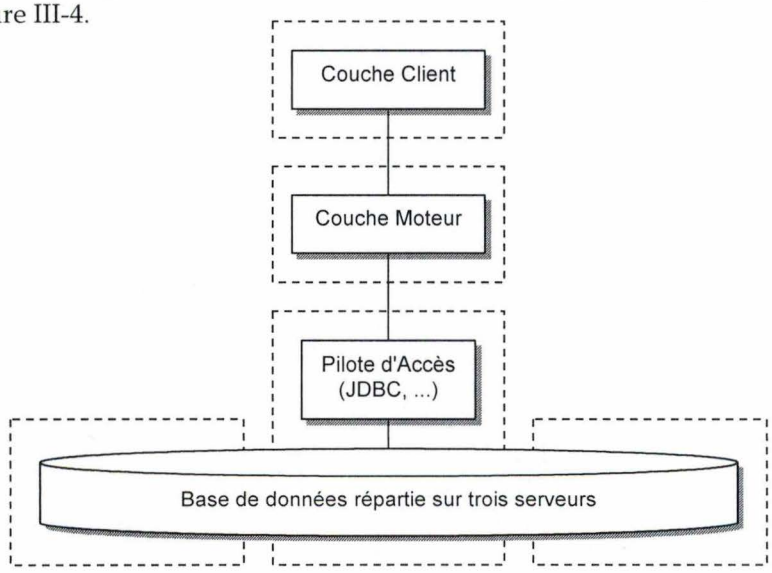

Figure III-4 Exemple de distribution des couches de l'architecture avec une base de données répartie sur trois serveurs

Du point de vue de notre architecture, la répartition de la base de données représente un souci mineur, puisqu'elle est rendue transparente par le pilote d'accès.

Notons que "client" et "serveur" définissent un rôle rempli par un environnement d'exécution donné dans une relation de collaboration avec d'autres environnements d'exécution. Ainsi, rien n'empêche la couche client et le moteur, présentés sur le schéma de la Figure III-4 comme étant sous le contrôle de deux environnements d'exécution distincts, de se localiser sur la même machine physique.

III.2 Couche client

La couche client met un langage à la disposition du développeur pour lui permettre de dialoguer avec la base de données sous-jacente. Ce dialogue est organisé autour du concept d'objet, dont l'anatomie est étudiée plus loin dans cette section (cfr III.2.2). Anticipativement, nous pouvons ici nous limiter à définir un objet comme un composant de l'architecture dont le rôle est d'encapsuler les enregistrements d'une table, à la manière d'un curseur. Sa structure est entièrement générique et adaptable à n'importe quel schéma de base de données. Il est dès lors naturel de retrouver deux classes de fonctionnalités: la manipulation des objets et la définition des objets.

III.2.1 Manipulation des objets

L'architecture procure un panel de fonctionnalités dédiées à la manipulation des objets et donc indirectement au traitement des enregistrements.

a) Opérations sur un objet

Les opérations définissent les manipulations de base dédiées au transfert de données à destination ou en provenance de la base de données.

- **Chargement.** Sur la base de la valeur d'un identifiant, un enregistrement est lu dans la base de données, et les valeurs de ses champs sont recopiées dans les structures mémoires de l'objet pourvues à cet effet. Après un chargement, les valeurs des champs de l'enregistrement sont identiques aux valeurs mémorisées par l'objet (pourvu qu'un autre utilisateur n'ait pas déjà modifié cet enregistrement).
- **Création.** Les valeurs contenues dans l'objet sont utilisées pour insérer un nouvel enregistrement dans la base de données. Après une création, les valeurs des champs du nouvel enregistrement sont identiques aux valeurs présentes dans l'objet, sauf pour les champs mis à jour automatiquement (comme les timestamps) par la base de données. Dans ce cas, un rechargement de l'objet peut être nécessaire.
- **Modification.** Les valeurs contenues dans l'objet sont utilisées pour modifier l'enregistrement préalablement chargé. Les suites d'une modification sont équivalentes à celles d'une création.
- **Suppression.** L'enregistrement préalablement chargé est supprimé de la base de données. Après une suppression, les valeurs contenues dans l'objet restent inchangées.
- **Réinitialisation.** L'objet est déconnecté de l'éventuel enregistrement préalablement chargé. Les valeurs contenues dans l'objet sont réinitialisées.
- **Changement des valeurs.** Les valeurs contenues dans l'objet peuvent être changées à tout instant. Ces valeurs pourront ensuite être utilisées dans le cadre d'une création ou d'une modification d'enregistrement.

Pour reprendre une terminologie des bases de données relationnelles, on pourrait comparer l'objet à un "curseur immobile". L'objet est en effet incapable de parcourir un ensemble d'enregistrements préalablement sélectionnés. Cette fonctionnalité est du ressort des requêtes que nous examinons plus loin (cfr III.2.1.d).

La résolution des conflits engendrés par les mises à jour concurrentes est assurée par l'objet. Nous avons opté pour la résolution a posteriori telle qu'elle a été décrite dans l'évaluation de JDBC (cfr II.3.2). Lors d'une modification ou d'une suppression, l'objet vérifie que l'enregistrement n'a pas été modifié entre-temps par un autre utilisateur. Dans l'affirmative, la modification ou la suppression échoue.

b) États d'un objet

On peut à tout moment s'informer sur l'état d'un objet. L'état d'un objet définit sa situation par rapport à un éventuel enregistrement sous-jacent et les valeurs qu'il contient. Un changement d'état est la conséquence de l'exécution d'une opération. Les états possibles sont les suivants:

- **Nouveau.** L'objet vient d'être créé. Il n'est connecté à aucun enregistrement. Les valeurs contenues dans l'objet sont initialisées et donc intactes.
- **Non-connecté et altéré.** Une ou plusieurs valeurs contenues dans un objet qui n'est connecté à aucun enregistrement ont été changées.
- **Réinitialisé.** L'objet et les valeurs de champs qu'il contient viennent d'être réinitialisés. L'objet n'est connecté à aucun enregistrement. Cet état est analogue à l'état "nouveau", mais fait suite à une opération de réinitialisation.
- **Chargé.** Cet état fait suite à une opération de chargement. L'objet vient d'être chargé à partir d'un enregistrement. Les valeurs contenues dans l'objet sont identiques à celles de l'enregistrement lu. L'objet est considéré comme connecté à cet enregistrement.
- **Connecté et altéré.** Une ou plusieurs valeurs contenues dans un objet connecté à un enregistrement ont été changées.
- **Créé.** Cet état fait suite à une opération de création. Les valeurs de champs de l'objet viennent d'être utilisées pour insérer un nouvel enregistrement dans la table sous-jacente. Elles sont identiques à celles de l'enregistrement créé, sauf dans certains contextes (cfr plus haut, description de l'opération de création). L'objet est alors connecté à cet enregistrement.
- **Modifié.** Cet état fait suite à une opération de modification. L'enregistrement auquel l'objet est connecté vient d'être mis à jour au moyen des valeurs contenues dans l'objet. L'objet reste connecté à cet enregistrement.
- **Supprimé.** Cet état fait suite à une opération de suppression. L'enregistrement auquel l'objet était connecté vient d'être supprimé. L'objet n'est plus connecté à aucun enregistrement.

La machine d'états finis de la Figure III-5 ci-dessous illustre les transitions autorisées entre les différents états que nous venons de décrire.

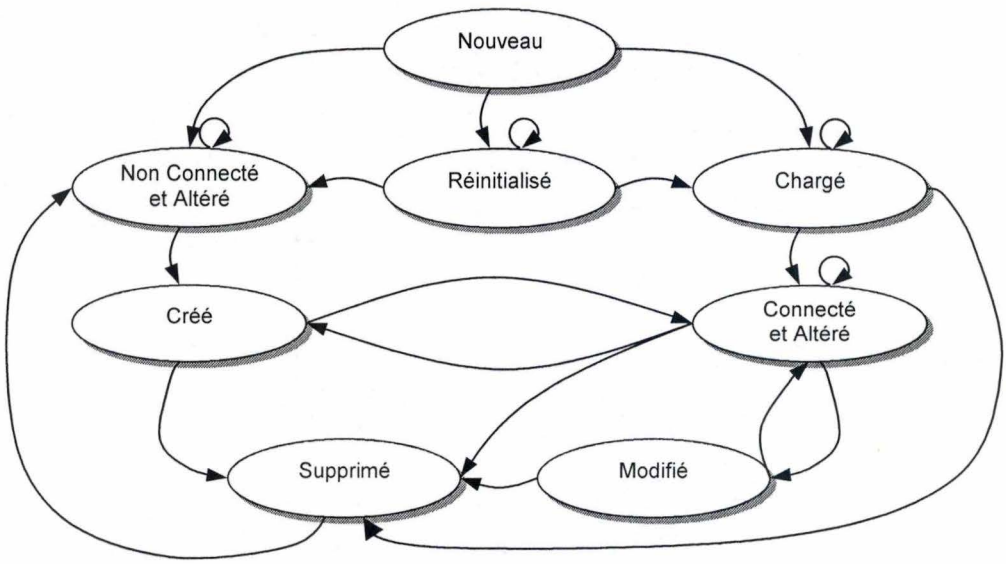

Figure III-5 Illustration des états et des transitions possibles.
Non exprimé sur le schéma: tous les états peuvent transiter vers les états *Réinitialisé* et *Chargé*.

Afin de ne pas alourdir le schéma, deux transitions ont été volontairement omises. En effet, les opérations de chargement et de réinitialisation sont autorisées à partir de n'importe quel état. Il faudrait donc une connexion vers les états *chargé* et *réinitialisé* en provenance de tous les autres états.
D'autre part, il faut remarquer que seuls les états *chargé* et *créé* permettent à un objet de se connecter à un enregistrement différent de celui auquel il était éventuellement déjà connecté. Les autres états résultent d'opérations traitant soit l'enregistrement courant, soit aucun enregistrement.

c) Navigation entre objets

Le modèle propose un système qui simule la navigation entre les objets. On distingue deux types de navigation:

- **Navigation vers "un", ou navigation simple**. A partir d'un enregistrement A, "naviguer vers un" signifie charger **un enregistrement B référencé par** A. La Figure III-6 illustre ce type de navigation en pseudo-code.

```
// Chargement d'une commande
COMMANDE.Charger où COMMANDE.ID = '123'

// Affichage du nom du client associé à la commande chargée
Afficher COMMANDE.CLIENT.NOM
```

Figure III-6 Exemple de navigation simple en pseudo-code

- **Navigation vers "plusieurs", ou navigation inverse.** A partir d'un enregistrement A, "naviguer vers plusieurs" signifie charger **l'ensemble des enregistrements B référençant** A. La Figure III-7 illustre ce type de navigation en langage pseudo-code.

```
// Chargement d'un client
CLIENT.Charger où CLIENT.ID = '345'

// Affichage des dates auxquelles le client chargé a passé une commande
Pour chaque CLIENT.COMMANDE:
  Afficher CLIENT.COMMANDE.DATE_COMMANDE
```

Figure III-7 Exemple de navigation inverse en pseudo-code

Dans l'exemple de la Figure III-7 ci-dessus, on constate en effet que l'on navigue vers un ensemble d'enregistrements (les commandes du client), contrairement à la navigation simple où l'on navigue vers un seul enregistrement (le client d'une commande).

d) Requêtes sur objets

Le rôle principal des requêtes est d'offrir la possibilité de parcourir un ensemble d'enregistrements sélectionnés sur une ou plusieurs tables, qui répondent à un certain nombre de critères, et suivant un ordre de parcours donné. De par sa nature, un objet seul ne peut remplir cette fonction: nous avons effectivement vu plus haut que l'objet est capable de manipuler les enregistrements exclusivement sur la base de leurs identifiants. Il n'est pas qualifié pour gérer un ensemble d'enregistrements.
Le modèle de ARROW fournit donc les structures et opérations nécessaires à la réalisation de requêtes simples ou complexes:

- **Association d'objets.** Les objets associés à une requête déterminent les tables et les champs impliqués dans le résultat. Le même objet ne peut être associé qu'à une seule requête à la fois. Par contre, un objet et une copie de cet objet peuvent être associés à la même requête, par exemple pour résoudre des références cycliques (un champ d'un enregistrement référençant un autre - voire le même - enregistrement de la même table).

- **Spécification de conditions.** Ces conditions s'appliquent à des champs encapsulés par des objets quelconques, c'est-à-dire associés ou non à la requête. Elles peuvent s'agréger afin de composer des conditions complexes. Les relations entre objets (c'est-à-dire l'équivalent des "joins") sont définies sous la forme de conditions.
- **Spécification de l'ordre de tri.** Les résultats de la requête peuvent être triés suivant un ordre paramétrable. L'ordre de tri est composé de champs encapsulés par des objets quelconques, c'est-à-dire associés ou non à la requête. L'ordre est croissant ou décroissant.
- **Obtention des résultats.** Il est possible de parcourir un par un tous les résultats de la requête. Chaque passage au résultat suivant déclenchera le garnissage des objets associés à la requête. La requête en tant que telle ne gère donc pas le stockage des valeurs des champs des enregistrements lus. Celles-ci sont affectées aux objets associés à la requête.
- **Comptage.** Il est possible d'obtenir le nombre d'enregistrements répondant aux conditions.
- **Position.** Il est possible d'obtenir le numéro d'ordre du résultat courant et de savoir si la requête est positionnée sur le dernier enregistrement de l'ensemble.

En outre, les requêtes procurent un ensemble d'opérations destinées à la mise à jour des données. Nous avons vu que les objets sont capables de manipuler un seul enregistrement à la fois. Pour des mises à jour ou suppressions multiples, il est alors obligatoire de traiter les enregistrements un par un, ce qui est plutôt lourd et non performant. Les requêtes apportent donc une réponse à cette limite grâce aux opérations suivantes:

- **Suppression.** Tous les enregistrements répondant aux critères de conditions de la requête sont supprimés de la base de données. Comme une même requête peut être associée à plusieurs objets (et donc différentes tables), et comme une opération de suppression est habituellement nécessaire sur une seule table à la fois, il faut spécifier sur lequel des objets la suppression doit être effectuée. Seuls les enregistrements de la table encapsulée par l'objet désigné pour la suppression seront supprimés.
- **Modification.** Tous les enregistrements répondant aux critères de conditions de la requête sont mis à jour avec les valeurs contenues dans un des objets de la requête. Tout comme pour la suppression, la modification ne peut agir que sur une seule table à la fois. Il est donc indispensable de désigner l'objet (et par là, la table) à mettre à jour. Seules les valeurs contenues dans cet objet, et qui ont été changées par le client, seront mises à jour dans la base de données.

Logiquement, l'opération de création ne trouve pas sa place ici puisque, par nature, elle manipule un et un seul enregistrement simultanément. Elle reste donc de la compétence de l'objet et n'est pas définie au niveau des requêtes.

e) Transactions

Toute opération sur les objets et requêtes impliquant ou non des mises à jour sur la base de données peut être explicitement validée ou annulée. Les opérations sur transactions sont décrites ci-dessous:

- **Association d'objets et requêtes.** Tous les objets ou requêtes peuvent faire l'objet d'une annulation ou validation. Bien que le nombre de transactions simultanées puisse être infini, le même objet (ou requête) ne peut être associé qu'à une seule transaction à la fois.
- **Annulation.** Lors d'une annulation, et les objets, et les requêtes, et la base de données doivent retrouver leur état initial, c'est-à-dire celui qu'ils avaient au démarrage de la transaction. On assure ainsi une synchronisation parfaite entre les valeurs stockées dans les structures mémoire des objets (et des requêtes) avec l'état de la base de données.
- **Validation.** La validation d'une transaction rend définitives et irréversibles les modifications apportées à la base de données à travers les objets associés à ladite transaction. Les valeurs stockées dans les objets (et les requêtes) associés restent inchangées.

Toute opération non effectuée dans le cadre d'une transaction est automatiquement validée et ne peut donc être annulée.

Une transaction démarre implicitement lors de sa création, ou immédiatement après une annulation ou une validation.

f) Gestion des erreurs

Toute erreur doit être interceptable et interprétable, qu'elle soit déclenchée par la base de données, par le langage de développement ou par notre modèle. Les messages d'erreurs doivent être clairs et, dans le cadre de la gestion d'une interface utilisateur, indiquer précisément la nature et la localisation du problème.

III.2.2 Définition des objets

Avant de pouvoir bénéficier des services offerts par un objet, de l'exploiter dans le cadre d'une navigation, d'une requête ou d'une transaction, il est nécessaire d'en définir la structure: quels sont les champs encapsulés, quels sont leurs types, ... Cette définition est en fait comparable à la conception d'un schéma de base de données virtuel. Qu'elle soit manuelle ou automatisée à l'aide d'un générateur, elle est plutôt du ressort d'un administrateur de bases de données que d'un programmeur.

a) Relation objet/enregistrement

Un objet est une représentation mémoire d'une partie de la structure et des données d'un enregistrement d'une même table, comme illustré dans la Figure III-8.

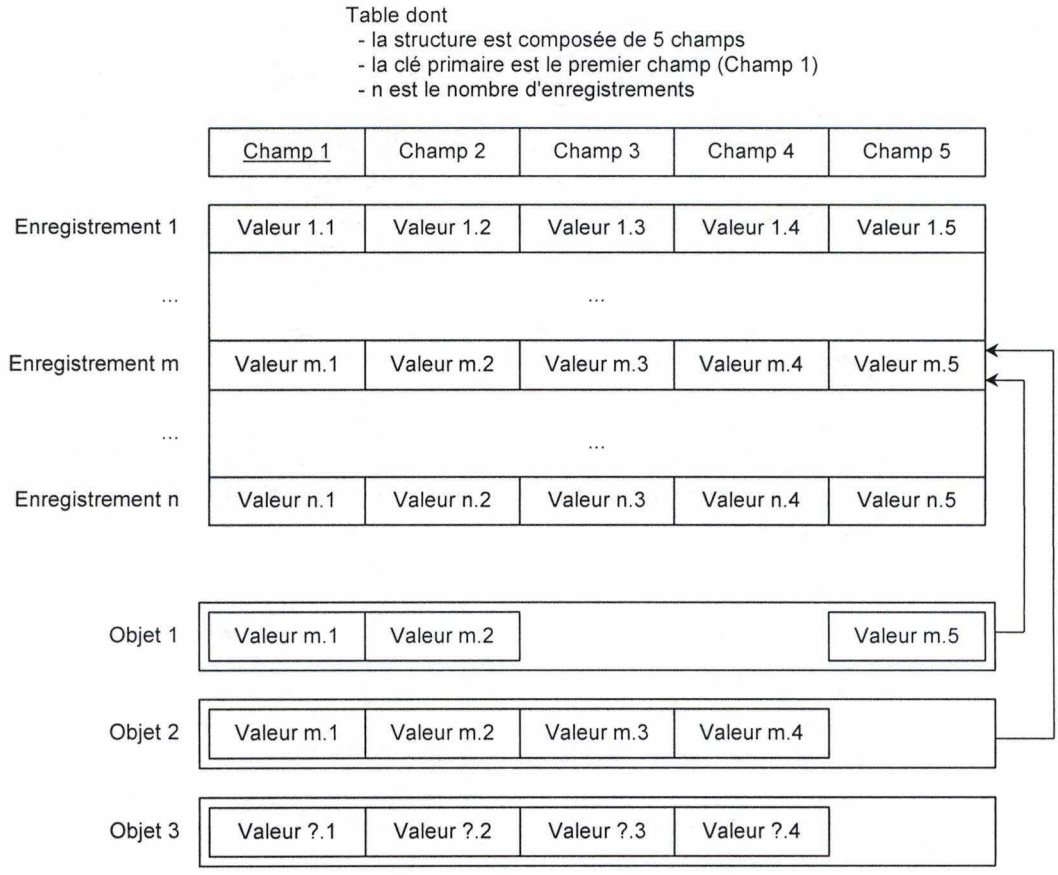

Figure III-8 Relation objet - enregistrement

De ce schéma découlent les observations suivantes:

- **Un objet est un masque de champs.** Un objet ne doit pas nécessairement encapsuler tous les champs de la table sous-jacente. Ainsi, *Objet 1* et *Objet 2* du schéma encapsulent chacun un ensemble différent de champs de la table. Pour ce faire, l'objet dispose de structures mémoire permettant la rétention des informations relatives aux champs encapsulés. Un objet ne peut manipuler que les champs qu'il encapsule. Il est garanti que les autres champs (les champs non encapsulés par cet objet) ne seront ni lus, ni modifiés, à moins qu'un autre objet ne les encapsule.
- **Le nombre d'objets est illimité.** On peut définir autant d'objets que l'on souhaite, quelle que soit la table sous-jacente et les champs encapsulés. Dans notre exemple, trois objets différents ont été définis, bien que *Objet 2* et *Objet 3* présentent la même structure.
- **La cardinalité de la relation "objet" - "enregistrement" est [0..*] - [0..1].** Autrement dit:
  - deux objets peuvent être connectés au même enregistrement (cfr *Objet 1* et *Objet 2*)
  - un objet n'est pas obligatoirement connecté à un enregistrement (cfr *Objet 3*)
  - un enregistrement peut n'être connecté à aucun objet (cfr *Enregistrements 1..m-1; m+1..n*).
- **Un objet est éphémère.** Un objet peut être détruit et créé à volonté sans incidence sur la base de données. Le cycle de vie d'un objet est donc totalement isolé de celui d'un enregistrement: un objet est volatile alors qu'un enregistrement est persistant, c'est-à-dire que la durée de vie d'un enregistrement s'étend au-delà de la durée de vie d'un objet.

b) Méta-données

La connexion d'un objet à un enregistrement implique inévitablement la présence d'un mécanisme de reconnaissance:

- **L'objet connaît les champs et la table qu'il encapsule.** L'objet dispose d'une méta-structure dont les composants permettent de retrouver les champs encapsulés ainsi que la table à laquelle ils appartiennent.
- **Les enregistrements d'une même table sont identifiables de manière univoque.** Seuls les champs de tables disposant d'une clé unique peuvent être encapsulés dans un objet. L'objet agrémente donc ses méta-structures d'informations permettant de retrouver les champs participant à la composition de la clé unique. Si une table dispose de plusieurs identifiants, on choisit de préférence un identifiant technique.

c) Contrôle d'intégrité

L'objet offre la possibilité de définir des règles de validité élémentaires (c'est-à-dire intra-enregistrement), telles que les domaines de valeurs acceptés et les champs requis. Ces règles sont appliquées aux valeurs de champs présents dans l'objet avant toute opération susceptible de modifier le contenu de la base de données sous-jacente. Il s'agit d'un contrôle de validité logique agissant préventivement aux contrôles d'intégrité physique déclenchés au sein de la base de données lors d'opérations de mises à jour. Ce contrôle préventif intra-enregistrement présente trois avantages principaux:

- **Performance.** La possibilité de valider les données localement (c'est-à-dire dans l'objet même, au niveau du client) avant de les envoyer vers la base de données optimise la réactivité de l'application cliente en cas d'erreur puisque, dans ce cas, ni le réseau, ni la base de données ne seront sollicités.
- **Conception.** Les règles de validité implémentées dans la base de données peuvent être peaufinées (fine-tuning) de manière logicielle. Ainsi, un champ non-requis au niveau de la base de données peut le devenir au niveau de l'objet. Un champ texte de longueur indéterminée dans la base de données peut se voir attribuer une limite (par exemple un maximum de 50 caractères) dans l'objet.

- **Détection.** Les erreurs détectées sont aisément localisables et identifiables sans que l'on doive interpréter un code d'erreur (souvent vague) en provenance de la base de données. Pour une interface utilisateur, la clarté des erreurs est un gage de qualité. Typiquement, les erreurs sont implémentées sous la forme d'exception.

Par ailleurs, aucune sémantique n'est prévue la gestion des règles d'intégrité extra-enregistrements. Ainsi, la gestion préventive de l'intégrité référentielle dans le cas d'une suppression nécessiterait de parcourir tous les enregistrements reliés à l'enregistrement à supprimer, avant d'avertir le client que l'opération ne peut être (le cas échéant) exécutée. Les structures et fonctionnalités inhérentes à ce type de traitement - ainsi qu'aux modifications et supressions en cascade - sont absentes de notre modèle et font donc l'objet d'une limite (cfr III.4).

Note: on remarque que les objets ne doivent pas nécessairement reproduire l'entièreté de la base de données sous-jacente car:

- seulement un ensemble de tables peuvent être encapsulées.
- un objet ne doit pas obligatoirement encapsuler tous les champs de la table cible.
- les domaines de valeurs peuvent être redéfinis dans l'objet, bien qu'il soit insensé de définir un domaine de valeurs moins restrictif au niveau de l'objet que celui défini dans la base de données.

d) Connexion au moteur

Les objets, requêtes et transactions sont les seuls composants de la couche client de notre modèle amenés à communiquer, par l'intermédiaire du moteur, avec une base de données. Ces composants doivent donc, à un moment donné, pouvoir désigner avec quelle base de données (quel que soit son type pourvu que le moteur le supporte) ils désirent dialoguer.
A cet effet, l'architecture doit disposer d'un service de connexion entre la couche client et le moteur permettant notamment de spécifier l'adresse de la base de données cible.
En pratique, seul l'objet devra disposer d'une connexion vers le moteur. La connexion d'une requête ou d'une transaction est implicitement disponible via les objets qui y sont associés. Certaines contraintes sont d'ailleurs liées aux connexions dans le cadre de l'utilisation des requêtes et transactions (cfr III.4.2).

III.3 Couche Moteur

Le moteur doit fournir les fonctionnalités nécessaires à l'exécution des ordres de manipulation de la base de données initiés par la couche client, et ce, de manière transparente pour le développeur.
Tout comme la couche client dispose d'un langage à l'attention du développeur, le moteur comporte une interface destinée à être exploitée par cette couche client. Le langage du moteur est a priori inaccessible au développeur et doit donc être considéré comme une boîte noire dont nous proposerons une modélisation en fin de travail (cfr V.2 et VI).

III.4 Limites et extensions potentielles

Puisqu'il est naturellement utopique de spécifier l'intégralité d'un modèle parfait (surtout dans le cadre de ce travail), des limites ont dû être posées. Chacune d'entre elles constitue une extension potentielle.

III.4.1 Gestion des données

- **Identifiants.** D'une part, seuls les champs d'une table possédant une clé unique constituée d'un ou plusieurs champs peuvent être encapsulés dans un objet, puisque celui-ci doit, à tout moment, pouvoir identifier l'enregistrement auquel il est connecté. D'autre part, au moins les champs participant à la composition de la clé unique doivent être encapsulés. Enfin, si une table dispose de deux identifiants, l'un d'entre eux devra être désigné comme identifiant d'enregistrement. On choisira de préférence un identifiant technique et surtout l'identifiant référencé par les clés étrangères des autres tables. En général, il s'agira de la clé primaire.
- **Intégrité.** Seule l'intégrité intra-enregistrement est prise en charge. Ainsi, l'intégrité extra-enregistrements (par exemple, l'intégrité référentielle et le contrôle des clés uniques) reste de la compétence de la base de données sous-jacente.
- **(Dé)blocages.** Aucune sémantique n'est prévue pour le blocage et le déblocage des enregistrements ("locking") .
- **Jointures.** Pour les requêtes de sélection, aucune sémantique n'est prévue pour l'équivalent des jointures gauches et droites ("left joins" et "right joins"). Seules les jointures classiques retrouvent un équivalent via les conditions applicables aux requêtes.

III.4.2 Gestion de la répartition

Nous avons vu plus haut (cfr III.2.2d)) que la couche client de notre architecture doit établir une connexion vers la couche moteur avant toute opération sur la base de données. Nous avons également examiné les différentes possibilités de répartition (cfr III.1.3). On peut donc constater qu'une même instance de couche client peut établir de multiples connexions vers de multiples couches moteur.

Dans un tel contexte, nous avons choisi de poser la limite suivante: une requête ou une transaction ne peut utiliser plus d'une connexion à la fois. En d'autres termes, puisque la connexion d'une requête ou transaction est déterminée par la connexion des objets associés, il est interdit d'associer deux objets ayant des connexions différentes à une même requête ou à une même transaction.

Sans cette limite, nous serions contraints d'implémenter un système de gestion de requêtes et transactions distribuées.

III.4.3 Gestion des événements

On pourrait envisager un processus de notification qui, lors d'une mise à jour ou suppression d'un enregistrement, permettrait à tous les objets connectés sur cet enregistrement d'être avertis. Deux approches sont possibles:

- **Définition de déclencheurs (triggers) dans la base de données.** Cette approche n'est malheureusement pas applicable puisque la couche inférieure de notre architecture utilise de préférence le modèle JDBC/SQL qui est incapable de capter le déclenchement de triggers.
- **Gestion d'une cache.** Le moteur de notre architecture pourrait gérer une cache qui conserverait une référence vers tous les objets et les enregistrements connectés. Lors de la modification d'un enregistrement, et pour autant qu'elle soit effectuée par la couche client, cette cache serait à même de

retrouver tous les objets connectés à cet enregistrement et donc de les informer de la mise à jour (ou de la suppression).

III.4.4 Générateur d'objets

Automatiser la génération d'objets (en fait, de classes) à partir du schéma de la base de données est une extension particulièrement intéressante. Nous avons vu plus haut que l'objet doit disposer de méta-informations, telles que la structure des champs qu'il encapsule (cardinalité, domaine de valeurs, ...) et les champs participants à la composition d'une clé unique. La génération de ces méta-informations est parfaitement automatisable à partir du schéma de la base de données sous-jacente. On peut par exemple l'imaginer sous la forme d'un outil permettant, entre autres, de sélectionner les tables et les champs à encapsuler, de choisir des domaines de valeurs plus restrictifs, ...

IV
Langage de manipulation de ARROW

Dans ce chapitre, nous allons décrire le langage de manipulation mis à la disposition du développeur. Ce langage fait partie intégrante de la couche client du modèle et est destiné à l'exploitation des bases de données. Le chapitre suivant étudiera le langage de définition qui permet de définir ses propres structures objet destinées à être exploitées par le langage de manipulation.

IV.1 Le noyau

IV.1.1 Coeur du noyau

Le noyau constitue le squelette de l'interface client de notre modèle. Il correspond à la notion d'objet que nous avons développée dans le chapitre précédent (cfr III.2). Il permet donc de définir et d'exploiter l'équivalent d'un "curseur immobile". Il se compose de trois classes abstraites comme nous l'illustre la Figure IV-1.

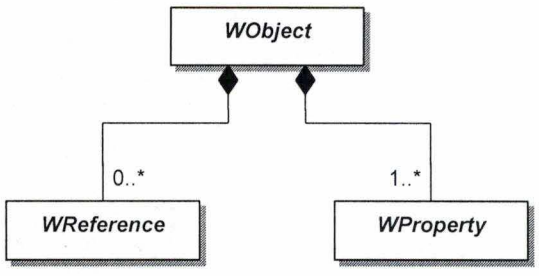

Figure IV-1 Noyau du modèle

Le noyau est une fondation permettant de représenter la totalité ou une partie (c'est-à-dire un ensemble restreint de tables et/ou de champs) de la structure d'une base de données sous une forme orientée objet. Il est dès lors naturel de retrouver une équivalence entre le monde relationnel et le monde orienté objet de notre modèle, comme nous le présente le Tableau IV-1.

Concept Relationnel	Equivalent dans le Noyau
champ: - nom - domaine de valeurs - cardinalité - valeur (d'une instance de champ)	*WProperty*
enregistrement: - nom - ensemble de champs - contraintes sur ces champs - ensemble de relations	*WObject*
relation: - nom - ensemble de champs participant à la relation - cardinalité	*WReference*

Tableau IV-1 Implémentation des concepts relationnels dans le noyau

Une instance de *WObject* est destinée à encapsuler un enregistrement par l'intermédiaire des instances de *WProperty* associées qui, elles, se chargent d'encapsuler les champs dudit enregistrement. *WReference* emballe la notion de relation dans un but de navigation.

On remarque que le nombre de classes est relativement limité (trois) et que, en tant que tel, le noyau ne peut reproduire tous les types de champs, tous les types de contraintes, etc... imaginables. Cet aspect justifie la nécessité de spécialisation et d'enrichissement de la sémantique du noyau.

IV.1.2 Extensions du noyau

En réponse au caractère abstrait du noyau, la Figure IV-2 propose un ensemble de classes concrètes dédiées - entre autres - à la gestion de domaines de valeurs particuliers. En pratique, un nombre indéfini de classes peut spécialiser *WProperty*. Nous nous sommes limités ici aux classes qui nous permettront de modéliser la base de données illustrative.

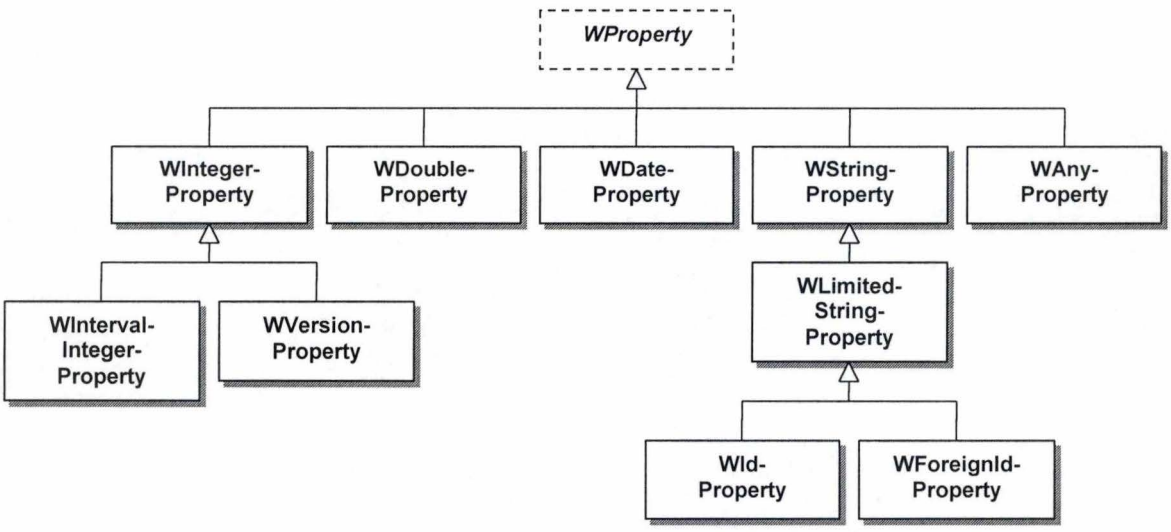

Figure IV-2 Spécialisations de *WProperty* proposées par le noyau

Chacune de ces classes sera étudiée plus loin (cfr IV.1.4). Néanmoins, il est important de déjà se familiariser avec la logique du modèle, principalement basée sur ses capacités de spécialisation. Le Tableau IV-2 décrit les domaines de valeurs de ces classes héritées de *WProperty*.

Classe	Domaine de valeurs des champs encapsulés
WIntegerProperty	Entier
WIntervalIntegerProperty	Entier compris dans un intervalle
WVersionProperty	Entier dédié à la gestion de la concurrence
WDoubleProperty	Réel
WDateProperty	Date
WStringProperty	Chaîne de caractères
WLimitedStringProperty	Chaîne de caractères de longueur comprise dans un intervalle
WIdProperty	Chaîne de caractères dédiée à la gestion des clés primaires
WForeignIdProperty	Chaîne de caractères dédiée à la gestion des clés étrangères
WAnyProperty	Type quelconque

Tableau IV-2 Domaines de valeurs des classes spécialisées de *WProperty*.

De manière analogue à *WProperty*, le modèle propose une spécialisation pour *WReference*, comme l'illustre la Figure IV-3. De nouveau, précisons que ces classes sont étudiées plus loin (cfr IV.1.6), mais sont présentées anticipativement pour nous permettre de modéliser la base de données illustrative.

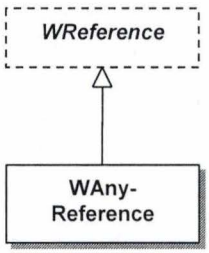

Figure IV-3 Spécialisations de *WReference*

La classe *WAnyReference* est à même de gérer la navigation d'un enregistrement à l'autre par l'intermédiaire d'instances de *WObject* et ce, quels que soient les champs participant aux clés étrangères et primaires qui définissent la relation entre lesdits enregistrements.

IV.1.3 Modélisation de la base de données illustrative

Nous sommes maintenant en possession de tous les éléments permettant la modélisation de la base de données illustrative.

Les champs définis par des domaines de valeurs équivalents et dont la sémantique est similaire sont encapsulés par une seule classe héritée de *WProperty*. Par exemple, les champs *PRODUIT.PRIX* et *DETAIL.PRIX* appartiennent tous les deux au domaine de valeurs réel et représentent le concept de prix. Nous avons donc une classe héritée de *WDoubleProperty* (*XPrixProperty*) qui prend en charge la gestion de ces deux champs. On procède de la même manière pour les autres champs. La Figure IV-4 illustre les classes ainsi définies.

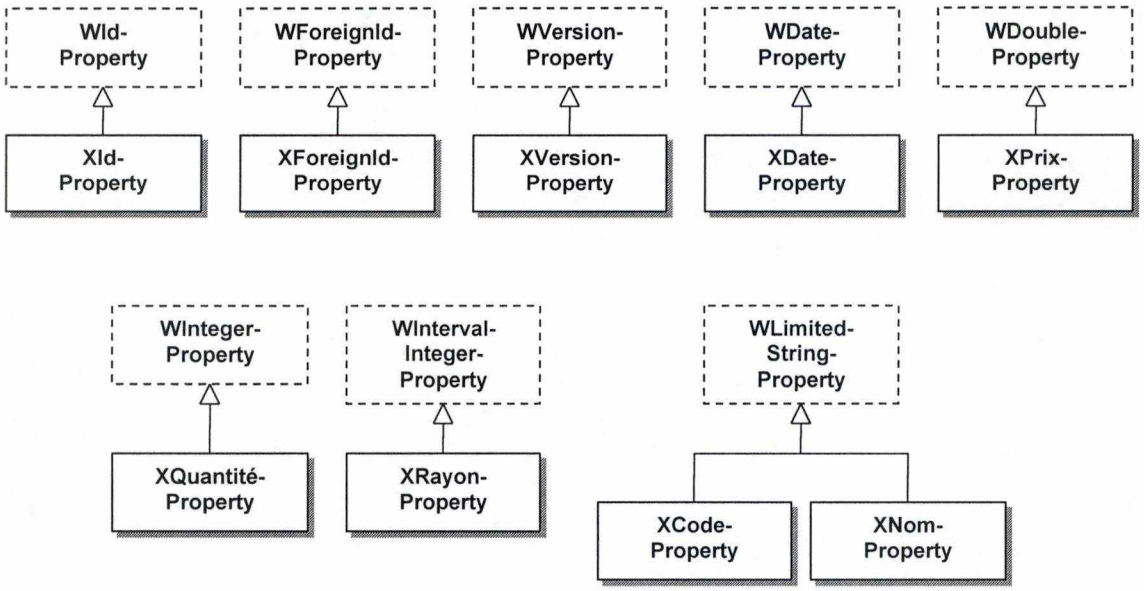

Figure IV-4 Spécialisations de *WProperty* spécifiques à la base de données illustrative

Notons que les classes spécifiques à la base de données illustrative sont préfixées par 'X' alors que celles du noyau sont préfixées par 'W'. Le tableau présente leurs domaines de valeurs.

Classe	Domaine de valeurs des champs encapsulés
XIdProperty	Clé primaire de 16 caractères
XForeignIdProperty	Clé étrangère de 16 caractères
XVersionProperty	Entier dédié à la gestion de la concurrence
XDateProperty	Date
XPrixProperty	Prix (réel)
XQuantitéProperty	Quantité (entier)
XRayonProperty	Numéro de rayon (entier compris entre 1 et 31 inclus)
XCodeProperty	Code de 0 à 10 caractères
XNomProperty	Nom de 0 à 50 caractères

Tableau IV-3 Domaines de valeurs des classes spécialisées de *WProperty* et spécifiques à la base de données illustrative

La Figure IV-5 présente une proposition de modélisation objet de la base de données illustrative. On y exploite les classes spécialisées de *WProperty* et *WReference* que nous venons d'examiner.
Puisque toutes les tables possèdent les champs *ID* et *VERSION*, nous définissons une classe mère (*XObject*) prenant en charge leur encapsulation. Ensuite, nous spécialisons quatre nouvelles classes, chacune d'elles correspondant à une des tables de la base de données illustrative.

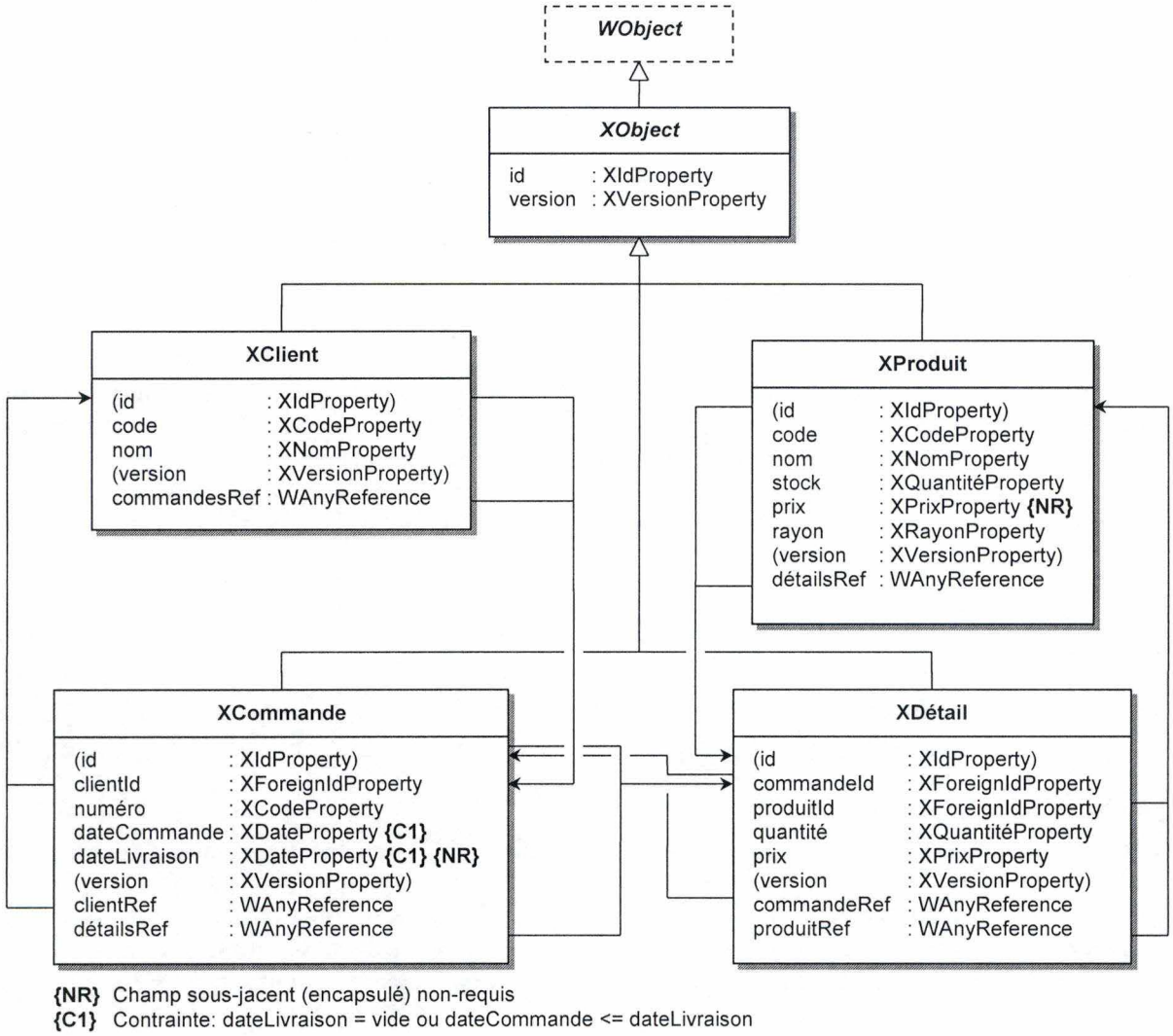

Figure IV-5 Modélisation de la base de données illustrative.
Note: Les attributs entre parenthèses sont implémentés par XObject.

Les flèches (⟶) expriment le concept de navigation. Il ne s'agit pas d'associations à proprement parler. Ainsi, une instance de *XCommande* ne contient aucune référence vers une instance de *XClient*, mais fournit la possibilité, par l'intermédiaire de l'attribut membre *clientRef*, de naviguer vers une instance de *XClient*. Nous verrons plus loin comment cela se concrétise (cfr IV.1.6.b). Remarquons également que les références dédiées à la navigation simple (vers un seul enregistrement) sont énoncées au singulier; les références consacrées à la navigation inverse (vers plusieurs enregistrements) sont exprimées au pluriel.
En outre, nous avons choisi ici de réutiliser telle quelle la classe *WAnyReference*. Une alternative aurait été de définir des classes spécialisées de *WReference* et spécifiques à la base de données illustrative: *WClientRef*, *WCommandesRef*, *WProduitRef*, etc ...

Les concepts de cardinalité, d'identifiant et de domaine de valeurs ne sont pas présentés sur le schéma. En effet, ceux-ci concernent les champs sous-jacents et pas les attributs membres. Ainsi, la classe *XCommande* ne pourrait se modéliser comme l'illustre la Figure IV-6.

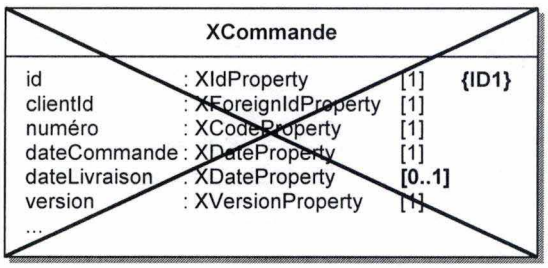

Figure IV-6 Modélisation incorrecte de la classe *XCommande*

Au sein de la classe *XCommande*, l'attribut *dateLivraison* a une cardinalité de [1]. Par contre, le champ encapsulé *DATE_LIVRAISON* possède, lui, une cardinalité de [0..1]. En outre, l'attribut *id* n'est pas identifiant de *XCommande*, mais bien identifiant de l'enregistrement sous-jacent.

Voyons maintenant comment les classes du noyau vont nous permettre d'exploiter la base de données illustrative sous sa forme objet.

IV.1.4 WProperty

Puisque le rôle de *WProperty* est d'encapsuler un champ d'un enregistrement, nous allons retrouver ici un ensemble de concepts familiers à la notion de champ.

a) Gestion des valeurs

WProperty permet de stocker temporairement (en mémoire) les valeurs de champs à destination ou en provenance des enregistrements de la base de données.
Des méthodes d'assignation et de consultation sont donc nécessaires pour pouvoir changer ou interroger la valeur conservée par une instance de *WProperty*. Lors d'une assignation, il est contrôlé si la nouvelle valeur appartient bien au domaine de valeurs. En cas d'échec, une exception est déclenchée. La gestion des domaines de valeurs proprement dits est prise en charge par les classes spécialisées de *WProperty*.

Modélisation
+ setValue(value : Any) > Exception
Stocke une valeur en mémoire ou déclenche une exception si cette valeur ne respecte pas le domaine de valeurs. L'état de l'instance courante de *WProperty* est changé à "altéré" via la méthode *setDirty()*.
Note: Le type Any signifie n'importe quel type de donnée, ce qui pose un problème en Java car les types primitifs ne sont pas assimilés à des objets. La solution est d'utiliser les objets standards Java qui emballent les types primitifs (par exemple java.lang.Integer pour les entiers).
+ getValue() : Any
Retourne une valeur préalablement stockée.

Par convention, les classes spécialisées de *WProperty* définissent leurs propres méthodes d'assignation et d'interrogation des valeurs. Ces méthodes offrent l'avantage de disposer d'un typage plus spécifique ("strong typing") que *setValue()* et *getValue()* de *WProperty*.

Modélisation spécifique à WIntegerProperty
+ set(value : Integer) > Exception
Idem *WProperty.setValue()*
+ get() : Integer
Idem *WProperty.getValue()*

Modélisation spécifique à WDoubleProperty
+ set(value : Double) > Exception
Idem *WProperty.setValue()*
+ get() : Double
Idem *WProperty.getValue()*

Modélisation spécifique à WDateProperty
+ **set(value : Date) > Exception**
  Idem *WProperty.setValue()*
+ **get() : Date**
  Idem *WProperty.getValue()*

Modélisation spécifique à WStringProperty
+ **set(value : String) > Exception**
  Idem *WProperty.setValue()*
+ **get() : String**
  Idem *WProperty.getValue()*

Modélisation spécifique à WAnyProperty
+ **set(value : Any) > Exception**
  Idem *WProperty.setValue()*
+ **get() : Any**
  Idem *WProperty.getValue()*

Note: On est contraint d'utiliser les dénominations *get()/set()* plutôt que *getValue()/setValue()*. En effet, la signature d'une méthode est déterminée uniquement par le nom de la méthode, le nombre et les types de paramètres. Le type de la valeur de retour n'intervient pas dans la signature et de plus, deux méthodes ayant la même signature ne peuvent avoir un type de valeur de retour différent. Nous ne pouvons donc utiliser *getValue()* pour retourner une valeur de type entier car *getValue()* tel qu'il est défini dans *WProperty* retourne déjà une valeur de type *Any*. Nous avons donc choisi *get()* en lieu et place de *getValue()*. Par souci d'uniformité, *set(value : int)* a été rajouté, bien que *setValue(value : int)* soit correct.

La Figure IV-7 présente quelques exemples de manipulation des valeurs, en exploitant les classes définies lors de la modélisation de la base de données illustrative (cfr plus haut, IV.1.3).

```
// Déclaration d'une variable de type XProduit.
XProduit produit = new XProduit();

// Assignation d'un nom
produit.nom.set("Eau minérale");

// Affichage du nom
System.out.println(produit.nom.get());

// Assignation d'un numéro de rayon en dehors de l'intervalle autorisé,
// c'est-à-dire [1,31]. Une exception sera déclenchée au run-time.
produit.rayon.set(50);

// Assignation d'une valeur numérique alors qu'une valeur de type
// texte est attendue. Le compilateur déclenchera une erreur.
produit.id.set(999);
```

Figure IV-7 Exemples de manipulation des valeurs avec *WProperty*

b) Etats

L'état d'une instance de *WProperty* détermine si la valeur stockée a été changée depuis sa précédente assignation par le moteur de ARROW, c'est-à-dire depuis la dernière opération réussie sur l'instance de *WObject* mère (cfr IV.1.5.a).
Lorsque l'on assigne une nouvelle valeur à une instance de *WProperty*, l'état devient implicitement "altéré"; après une opération réussie sur l'instance de *WObject* mère, l'état est automatiquement réinitialisé à "non-altéré". En guise d'optimisation, *WObject* ne traite, lors d'une mise à jour, que les instances de *WProperty* dont la valeur a été changée, c'est-à-dire dont l'état est "altéré".

Modélisation
+ setDirty(dirty : Boolean)
Définit la valeur comme "altérée" ou "non-altérée"
+ isDirty() : Boolean
Indique si la valeur a été altérée depuis sa réinitialisation à "non-altéré"

La Figure IV-8 présente quelques exemples de manipulation des états.

```
// Déclaration d'une variable de type XProduit.
XProduit produit = new XProduit();

// Affichage de 'false' car aucune valeur n'a encore été assignée.
System.out.println(produit.nom.isDirty());

// Assignation d'une valeur
produit.nom.set("Whisky");

// Affichage de 'true' puisque l'on vient d'assigner une valeur.
System.out.println(produit.nom.isDirty());
```

Figure IV-8 Exemples de manipulation des états avec *WProperty*

c) Méta-données

Les métas-données ayant servi à la création d'une instance de *WProperty* sont accessibles en consultation. Elles seront décrites en détails dans le chapitre suivant, conjointement à l'étude du langage de définition de ARROW. Elles permettent au développeur de disposer des informations suivantes:

- **Instance de WObject mère**. Permet de retrouver l'instance de *WObject* (cfr IV.1.5) mère associée.
- **Cardinalité**. Détermine s'il est obligatoire (cardinalités [1-1]) ou facultatif (cardinalités [0-1]) de fournir une valeur de champ à une instance de *WProperty*.
- **Identifiant**. Indique si le champ encapsulé par une instance de *WProperty* participe à l'index unique utilisé pour connecter l'instance de *WObject* mère à l'enregistrement sous-jacent.
- **Cachet ("stamp" ou estampille)**. Indique si le champ encapsulé par une instance de *WProperty* est un cachet dédié à la gestion de la concurrence.
- **Domaine de valeurs**. Une classe héritée de *WProperty* peut fournir un ensemble d'informations relatives à son domaine de valeurs.
- **Informations physiques**. Fournit le nom physique du champ encapsulé. Peut par exemple être utile pour le débogage.
- **Informations logiques**. Fournit un nom logique et une description succincte destinés à l'utilisateur final.

Ces données peuvent être exploitées par le développeur pour concevoir des interfaces utilisateur proactives et dynamiques, avec une mise en évidence des champs requis, un affichage paramétrique des libellés de zones et des domaines de valeurs, ...

Modélisation
+ getParent() : WObject
Retourne l'instance de *WObject* associée (l'instance de *WObject* mère)
+ isRequired() : Boolean
Indique si l'instance courante de *WProperty* requiert une valeur.
+ isKey() : Boolean
Indique si le champ encapsulé participe à la composition de l'identifiant.
+ isStamp() : Boolean
Indique si le champ encapsulé est un cachet.
+ getFieldName() : String
Retourne le nom physique du champ encapsulé.
+ getLabel() : String
Retourne l'étiquette (nom logique) de l'instance courante de *WProperty*.

+ getDescription() : String
Retourne la description fonctionnelle (en termes compréhensibles par l'utilisateur).

Modélisation spécifique à WIntervalIntegerProperty
+ getMin() : Integer
Retourne la borne inférieure de l'intervalle
+ getMax() : Integer
Retourne la borne supérieure de l'intervalle

Modélisation spécifique à WLimitedStringProperty
+ getMin() : Integer
Retourne le nombre de caractèree minimum requis
+ getMax() : Integer
Retourne le nombre de caractères maximum autorisés

La Figure IV-9 présente des exemples d'utilisation des métas-données.

```
// Déclaration d'une variable de type XProduit.
XProduit produit = new XProduit();

// Cardinalité: affichage de 'true' et puis 'false'
System.out.println(produit.nom.isRequired());
System.out.println(produit.prix.isRequired());

// Identifiant: affichage de 'true' et puis 'false'
System.out.println(produit.id.isKey());
System.out.println(produit.nom.isKey());

// Domaine de valeurs: affichage de '16' et puis '50'
System.out.println(produit.id.getMin());
System.out.println(produit.nom.getMax());

// Champ physique: affichage de 'STOCK'
System.out.println(produit.stock.getFieldName());
```

Figure IV-9 Exemples d'exploitation des métas-données avec *WProperty*

d) Modélisation

Maintenant que nous avons abordé toutes les fonctionnalités attendues de *WProperty*, nous pouvons présenter la modélisation de son interface publique, comme l'illustre la Figure IV-10 ci-dessous:

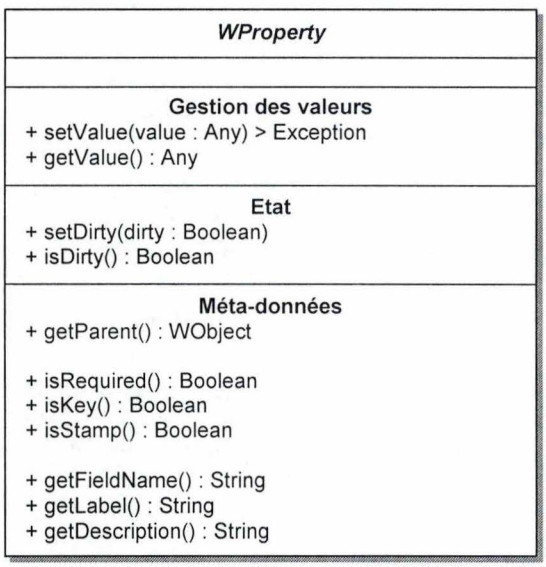

Figure IV-10 Modélisation de *WProperty*

IV.1.5 WObject

Une instance de *WObject* est destinée à emballer un ensemble de champs appartenant à un enregistrement issu d'une et une seule table de la base de données sous-jacente. La même instance de *WObject* ne peut donc encapsuler des champs provenant de tables différentes. Par contre, elle est tout à fait capable d'encapsuler les champs d'une vue (view) qui elle-même inclut des champs de plusieurs tables. Ce sont les instances de *WProperty* associées qui déterminent les champs à encapsuler.

Il est important de percevoir qu'une instance de *WObject* n'est pas un enregistrement, mais plutôt une image (tampon) d'enregistrement. Contrairement à un curseur, une instance de *WObject* ne peut voyager d'un enregistrement à l'autre parmi un ensemble. Elle manipule un enregistrement seulement à partir de son identifiant. Il n'est donc possible que de charger, créer, modifier ou supprimer un seul enregistrement à la fois. Les manipulations sur plusieurs enregistrements sont du ressort de la classe *WCollection* (cfr IV.2.3).

De plus, puisqu'une instance de *WObject* ne doit pas nécessairement encapsuler tous les champs de la table sous-jacente, on peut comparer *WObject* à un masque de champs. En outre, la définition de plusieurs *WObject* sur la même table est possible, voire utile, par exemple dans des contextes où seul un nombre restreint de champs doivent être manipulés.

Même si *WObject* ne contient aucune méthode abstraite, nous avons choisi de la déclarer comme telle, afin d'en forcer la spécialisation.

a) Opérations

Les opérations représentent le centre névralgique de *WObject*. Elles permettent de manipuler les enregistrements à destination ou en provenance de la base de données, c'est-à-dire, entre autres, de charger créeer, modifier ou supprimer.
Avant toute opération de modification d'enregistrement, il est vérifié que les valeurs stockées dans les instances de *WProperty* associées respectent bien les règles d'intégrité intra-enregistrement.
Remarquons que les règles d'intégrité intra-champ, comme les types et domaines de valeurs, ne doivent plus être recontrôlées, puisqu'elles ont déjà été vérifiées lors de l'assignation des valeurs de champs aux instances de *WProperty*. Il est dès lors impossible de trouver une valeur de champ non valide dans une instance de *WProperty*.
Par contre, le contrôle des règles impliquant un groupe d'instances de *WProperty* intervient au niveau de *WObject*. Par défaut, seule la cardinalité des champs est vérifiée. De nouvelles classes héritées de *WObject* peuvent spécialiser ce contrôle d'intégrité pour gérer d'autres règles comme l'exclusion (par exemple lorsqu'un champ peut être garni seulement si un autre est vide), la dérivation (par exemple lorsque la valeur d'un champ est fonction de la valeur d'autres champs), ...

Modélisation

+ load() > Exception
Charge un enregistrement à partir des valeurs de champs des instances de *WProperty* participant à la composition de l'identifiant. L'état des instances de *WProperty* associées est mis à "non-altéré". Une exception est déclenchée si aucun ou plus d'un enregistrement est trouvé.

+ load(condition : WCondition) > Exception
Charge un enregistrement à partir d'une condition. Les conditions sont étudiées plus loin (cfr IV.2.1). L'état des instances de *WProperty* associées est mis à "non-altéré". Une exception est déclenchée si aucun ou plus d'un enregistrement répond à la condition.

+ reload() > Exception
Recharge l'enregistrement auquel on est connecté (c'est-à-dire l'enregistrement préalablement chargé ou créé, même si une modification a été effectuée depuis) à partir des valeurs de champs des instances de *WProperty* participant à la composition de l'identifiant telles qu'elles étaient lors de la dernière opération réussie. L'état des instances de *WProperty* associées est mis à "non-altéré". Déclenche une exception si aucun ou plus d'un enregistrement est trouvé.

+ create() > Exception
Crée un enregistrement dans la base de données à partir des valeurs de champs des instances de *WProperty* associées. Préalablement, un contrôle d'intégrité est effectué et une exception est déclenchée si le contrôle échoue. L'état des instances de *WProperty* associées est mis à "non-altéré". Une exception est également déclenchée si la création est refusée par la base de données.

+ modify() > Exception
Modifie dans la base de données l'enregistrement auquel on est connecté. L'enregistrement doit avoir été préalablement chargé ou créé (même si une autre modification a été effectuée depuis).
Seules les valeurs de champs changées dans les instances de *WProperty* (cfr la méthode *isDirty()* de *WProperty*) sont modifiées dans la base de données. Un contrôle d'intégrité est préalablement effectué et une exception est déclenchée si le contrôle échoue. L'état des instances de *WProperty* associées est mis à "non-altéré". Une exception est également déclenchée si aucune valeur de champ d'instance de *WProperty* n'a été changée par le client ou si la modification est refusée par la base de données.

+ save() > Exception
Si on est connecté à un enregistrement, cette méthode essaie d'abord d'effectuer une modification via la méthode *modify()* suivi par une création via la méthode *create()* en cas d'échec de la modification. Sinon, une création est d'abord tentée, suivie par une modification si la création échoue. Les exceptions déclenchées sont identiques à celles des méthodes *create()* et *modify()*.

+ remove() > Exception
Supprime de la base de données l'enregistrement auquel on est connecté. L'enregistrement doit avoir été préalablement chargé ou créé (même si une modification a été effectuée depuis). Une exception est déclenchée si la suppression est refusée par la base de données.

+ revert()
Restitue un enregistrement, c'est-à-dire réinitialise les valeurs de champs des instances de *WProperty* associées avec les valeurs de champs des instances de *WProperty* telles qu'elles étaitent lors de la dernière opération réussie. Aucun accès à la base de données n'est effectué.

+ setNull()
Assigne toutes les valeurs de champs des instances de *WProperty* associées à vide. Aucun accès à la base de données n'est effectué.

+ reset()
Réinitialise l'instance de *WObject* courante et les instances de *WProperty* associées à un état identique à l'état initial faisant suite à la construction de ces objets. Aucun accès à la base de données n'est effectué.

La Figure IV-11 présente quelques exemples d'utilisation des opérations.

```
// Déclaration d'une variable de type XProduit.
XProduit produit = new XProduit();

// Chargement d'un produit à partir d'un identifiant que l'on
// suppose existant.
produit.id.set("TG5TEGD6ER4RTE6D");
produit.load();

// Chargement d'un autre produit à partir de son nom
produit.load(new WIsEqualToValue(produit.nom, "Pipe"));

// Affichage du stock disponible pour le produit chargé
System.out.println(produit.stock.get());

// Changement du nombre de produits en stock
produit.stock.set(produit.stock.get() + 10);

// Sauvegarde dans la base de données
produit.modify();
```

Figure IV-11 Exemples d'exploitation des opérations de *WObject*

b) Etats

WObject fait la distinction entre deux types d'états:

- **Etat opérationnel.** Cet état informe sur la dernière opération effectuée sur une instance de *WObjet* c'est-à-dire:
  - "nouveau" si l'instance vient d'être créée
  - "chargé" si la dernière opération effectuée est un chargement
  - "créé" si la dernière opération effectuée est une création

- "modifié" si la dernière opération effectuée est un remplcament
- "supprimé" si la dernière opération effectuée est une suppression
- "réinitialisé" si la dernière opération effectuée est une réinitialisation

Note: Une opération de restitution n'affecte pas l'état.

- **Etat dérivé.** Ce type d'état est "calculable" à partir soit de l'état opérationnel de l'instance de *WObject*, soit à partir de l'état des instances de *WProperty* associées. Les états dérivés suivants sont disponibles:
  - "connecté" si l'état opérationnel est chargé, créé ou modifié
  - "altéré" si au moins une valeur de champ de l'une des instances de *WProperty* est à l'état "altéré"

On retrouve en fait les états décrits dans les spécifications du modèle (cfr III.2.1.b), mais sous une autre forme puisque ici, nous avons scindé les états "connecté" et "altéré" des états opérationnels.

Modélisation
+ getStatus() : WObjectStatusEnumeration
Retourne l'état opérationnel de l'instance courante, suite à l'exécution réussie d'une opération.
Le type *WObjectStatusEnumeration* contient les valeurs {*new*, *loaded*, *created*, *modified*, *removed*, *reset*}.
+ isBinded() : Boolean
Informe si l'instance courante est connectée à un enregistrement.
+ isDirty() : Boolean
Informe si au moins une des valeurs de champs des instances de *WProperty* a un état "altéré" (via la méthode *isDirty()* de *WProperty*).

c) Méta-données

Les informations relatives à la structure d'une instance de *WObject* sont accessibles en lecture. Celles-ci peuvent par exemple être utilisées dans le cadre de la construction dynamique d'interfaces utilisateur. On retrouve les métas-données suivantes:

- **Instances de *WProperty* associées.** La liste des instances de *WProperty* associées est interrogeable à l'exécution.
- **Instances de *WReference* associées.** La liste des instances de *WReference* associées est également disponible en consultation.
- **Informations physiques.** Le nom physique de la table encapsulée est mis à la disposition du développeur (pour par exemple aider au débogage).
- **Informations logiques.** D'autres informations plutôt destinées à l'utilisateur final sont également disponibles.

Modélisation
+ getProperty(n : int) : WProperty > Exception
Retourne la *n*ième instance de *WProperty* associée, ou déclenche une exception si *n* dépasse le nombre d'instances de *WProperty* associées.
+ countProperties() : Integer
Retourne le nombre d'instances de *WProperty* associées.
+ getReference(n : int) : WReference > Exception
Retourne la *n*ième instance de *WReference* associée, ou déclenche une exception si *n* dépasse le nombre d'instances de *WReference* associées.
+ countReferences() : Integer
Retourne le nombre d'instances de *WReference* associées.
+ getTableName() : String
Retourne le nom physique de la table encapsulée par l'instance de *WObject* courante.
+ getLabel() : String
Retourne un nom logique représentant l'instance de *WObject* courante.
+ getDescription() : String
Retourne la description fonctionnelle de l'instance de *WObject* courante

La Figure IV-12 présente des exemples d'utilisation des métas-données.

```
// Déclaration d'une variable de type XProduit.
XProduit produit = new XProduit();

// Affichage du nombre d'instances de WProperty associées,
// c'est-à-dire '7'.
System.out.println(produit.countProperties());

// Affichage du nom de la table sous-jacente, c'est-à-dire 'PRODUIT'
System.out.println(produit.getTableName());
```

Figure IV-12 Exemples d'exploitation des métas-données avec *WObject*

d) Modélisation

La Figure IV-13 ci-dessous résume la modélisation de *WObject*.

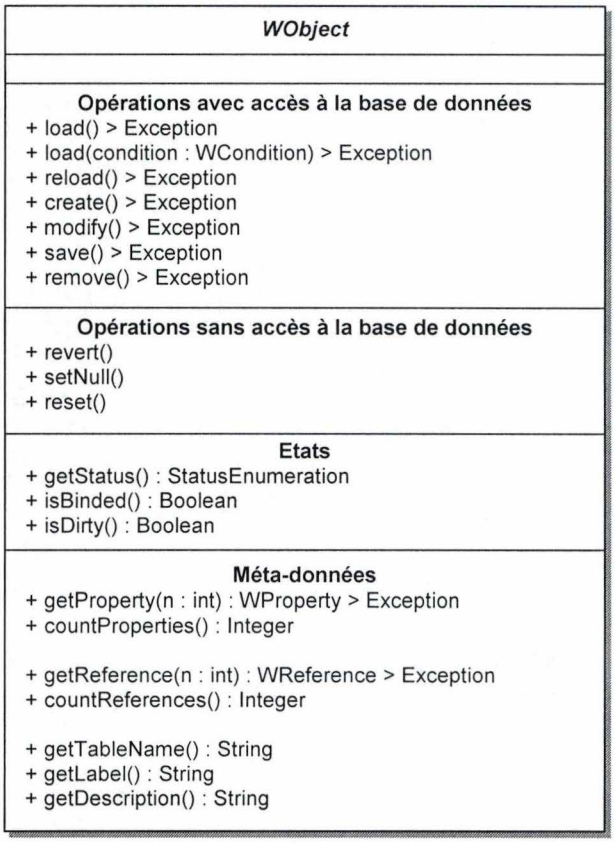

Figure IV-13 Modélisation de *WObject*

IV.1.6 WReference

WReference est une classe abstraite qui pose les bases nécessaires à la navigation entre enregistrements. Les deux types de navigation sont gérés: simple et inverse. Malgré le fait que *WReference* exploite diverses structures qui n'ont pas encore été présentées telles que *WCondition* (cfr IV.2.1) et *WCollection* (cfr IV.2.3), nous avons choisi de l'étudier à cet endroit du travail puisqu'il s'agit d'une classe faisant partie intégrante du noyau.

Dans un processus de navigation, deux instances de *WObject* entrent en jeu: l'instance source et l'instance cible. L'instance source est en fait l'instance de *WObject* à laquelle l'instance de *WReference* est associée par construction. L'instance cible doit être explicitement fournie lors de la navigation: *WRefe-*

rence ne prend pas en charge la création des instances cibles, mais il est tout à fait imaginable d'implémenter cette fonctionnalité dans une classe spécialisée.
Avant toute opération de navigation, une instance de *WReference* vérifie au préalable que l'instance cible est valide et utilisable. Par exemple, l'instance de *WReference XCommande.clientRef* n'accepte de naviguer que vers des instances de *XClient*. Une fois l'instance cible déclarée valide, il faut définir les conditions qui en permettront le chargement à partir de la base de données. Typiquement, ces conditions sont constituées de clauses d'égalités entre les champs identifiants de l'instance cible et les valeurs de champs (normalement des clés étrangères, dans le cas cas d'une navigation simple) correspondantes dans l'instance source.

a) Méta-données

Les méta-données de *WReference* se limitent à fournir l'instance de *WObject* mère associée.

Modélisation
+ getParent() : WObject
Retourne l'instance de *WObject* associée (l'instance de *WObject* mère)

b) Navigation simple

La navigation simple permet de charger un enregistrement dans une instance de WObject cible à partir des valeurs de champs composant la clé étrangère d'une instance source. Il est ainsi possible de charger un client à partir d'une commande grâce à la condition implicite suivante: *CLIENT.ID = COMMANDE.CLIENT_ID*.
On navigue donc vers un enregistrement référencé par l'enregistrement courant.

Modélisation
+ getTarget(target : WObject) : WObject > Exception
Charge l'objet cible fourni en paramètre et retourne cet objet cible pour permettre sa réutilisation immédiate dans la même ligne de code. Une exception est retournée si l'instance de *WObject* cible est non valide ou si aucun ou plus d'un enregistrement cible est trouvé.

La Figure IV-14 illustre des exemples de navigation simple.

```
// Déclaration de variables de type XCommande et XClient
XCommande commande = new XCommande();
XClient   client   = new XClient();

// Chargement d'une commande à partir d'un identifiant que l'on
// suppose existant.
commande.id.set("HYGET5DFKNBGDVCF");
commande.load();

// Navigation vers le client associé à la commande
commande.clientRef.getTarget(client);

// Affichage du nom du client
System.out.println(client.nom.get());
```

Figure IV-14 Exemples de navigation simple

c) Navigation inverse

Grâce à la navigation inverse, il est possible de sélectionner un ensemble d'enregistrements dans une instance cible de *WCollection* à partir des valeurs de champs composant la clé primaire d'une instance source. Il est ainsi possible de charger les commandes à partir d'un client grâce à la condition implicite suivante: COMMANDE.*CLIENT_ID* = *CLIENT.ID*.

On navigue donc vers les enregistrements qui référencent l'enregistrement courant. Le fait que la navigation vers la cible peut mener à plusieurs enregistrements, nous impose d'utiliser *WCollection* (cfr IV.2.3) pour en permettre le parcours.

Modélisation

+ getTarget(targetCollection : WCollection, targetObject : WObject) : WCollection > Exception

Affecte les conditions nécessaires ainsi que l'instance de *WOject* cible à une instance de *WCollection* pour permettre le parcours de tous les enregistrements cible au moyen de cette instance de *WCollection*. Après l'appel à cette méthode, il est naturellement possible de créer d'autres conditions, d'associer d'autres instances de *WObject*, ou de spécifier un ordre de parcours particulier à l'instance de *WCollection* retournée. Ensuite, la lecture des enregistrements proprement dite peut être initiée (via la méthode *next()* de *WCollection*).

Une exception est retournée pour les mêmes raisons que celles évoquées dans la navigation simple.

+ getTarget(targetCollection : WCollection) : WCollection > Exception

Cette méthode est fonctionnellement identique à la précédente, mais on ne spécifie ici aucune instance de *WObject* cible. En fait, la méthode essaie de trouver une instance cible parmi des instances de *WObject* déjà associées à l'instance de *WCollection* passée en paramètre. Une exception est déclenchée si une telle instance ne peut être trouvée.

La Figure IV-15 illustre des exemples de navigation inverse.

```
// Déclaration de variables de type XCommande, XClient et WCollection
XCommande   commande = new XCommande();
XClient     client   = new XClient();
WCollection col      = new WCollection();

// Chargement d'un client à partir d'un identifiant que l'on
// suppose existant.
client.id.set("SDDGTRE5HDYFERES");
client.load();

// Navigation vers les commandes associées au client
client.commandesRef.getTarget(col, commande);

// Affichage des dates de commande
while(col.next()
  System.out.println(commande.dateCommande.get());
```

Figure IV-15 Exemples de navigation inverse

d) Modélisation

La Figure IV-10 ci-dessous résume la modélisation de *WReference*:

WReference
Méta-données + getParent() : WObject
Navigation simple + getTarget(target : WObject) : WObject > Exception
Navigation inverse + getTarget(targetCollection : WCollection, targetObject : WObject) : WCollection > Exception + getTarget(targetCollection : WCollection) : WCollection > Exception

Figure IV-16 Modélisation de *WReference*

IV.2 Collections

Les collections offrent les fonctionnalités équivalentes à celles des requêtes telles que nous les avons décrites dans les spécifications du modèle (cfr III.2.1.d). Elles permettent la sélection dans la base de données de plusieurs enregistrements issus de différentes tables, répondant à un ensemble de conditions et suivant un ordre donné, tout en exploitant au maximum les composants que nous avons définis dans le noyau. La règle du jeu étant toujours de s'inscrire dans un contexte dénué de tout SQL ou autre langage d'accès aux bases de données.

La Figure IV-17 illustre les relations existantes entre les classes coopérant à la gestion des collections. *WObject* a déjà été étudié plus haut (cfr IV.1.5). Les trois autres classes sont examinées dans cette section.

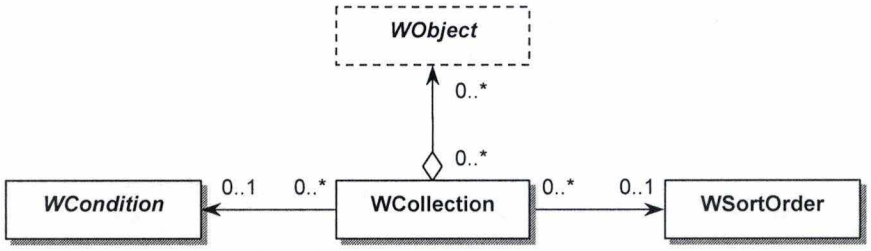

Figure IV-17 Classes intervenant dans la gestion des collections

Avant d'aborder la notion de collection en tant que telle, nous allons auparavant définir deux composants essentiels: les conditions (*WCondition*) et les ordres de tri (*WSortOrder*).

IV.2.1 WCondition

WCondition est la classe de base d'un ensemble de classes dédiées à la gestion des conditions. Ici encore nous devons porter une attention toute particulière à l'extensibilité pour ne pas être contraint de proposer un modèle définissant tous les types de conditions imaginables.

De quels fonctionnalités devons nous disposer pour modéliser un langage de conditions orienté objet de base?

- Opérateurs de comparaison classiques: =, <>, <, >, <=, >= et *pattern matching*
- Opérateur d'inversion de condition: *not*
- Opérateurs d'agrégation de conditions: *and* et *or*
- Comparaison d'un champ avec une valeur au moyen d'un opérateur quelconque
- Comparaison d'un champ avec une liste de valeurs au moyen de l'opérateur d'égalité ou de différence
- Comparaison d'un champ avec un autre champ, même appartenant à une autre table, au moyen d'un opérateur quelconque

De ces fonctionnalités, on peut tirer les observations suivantes, qui nous serviront de support pour la conception du modèle:

- Quel que soit le type de condition, il est toujours possible d'en inverser le sens (via l'opérateur *not*). Cette fonctionnalité est donc intégrée à la classe de base *WCondition*.

- Deux grands types de conditions se distinguent: les agrégations de conditions et les conditions de comparaison proprement dites. Cette deuxième catégorie se divise en deux sous-catégories: les conditions de comparaison d'un champ avec une ou plusieurs valeurs et les conditions de comparaison d'un champ avec un autre champ.
- Un champ est impliqué dans une condition sous sa forme object, c'est-à-dire via une instance de *WProperty* qui l'encapsule. Donc, la comparaison d'un champ avec une valeur requiert l'intervention d'une seule instance de *WProperty*, tandis que la comparaison d'un champ avec un autre champ en requiert deux.

La Figure IV-18 sur la page suivante expose le diagramme des classes dédiées à la gestion des conditions. Pour proposer un meilleur aperçu de la répartition des fonctionnalités à travers les différentes classes, les méthodes sont présentées conjointement aux classes.

On remarque que les classes traitant les opérateurs < et > pourraient partager le caractère strict de la comparaison par le biais d'une classe mère commune. Toutefois, cette option n'aurait apporté aucune amélioration fonctionnelle.

De plus, de manière similaire à *WProperty*, nous aurions pu également spécialiser les classes filles de *WSimpleValueCondition* pour assurer la gestion typées des valeurs. Par exemple, une classe *WIsEqualToInteger* (spécialisée de *WIsEqualToValue*) accepterait seulement les valeurs de type entier.

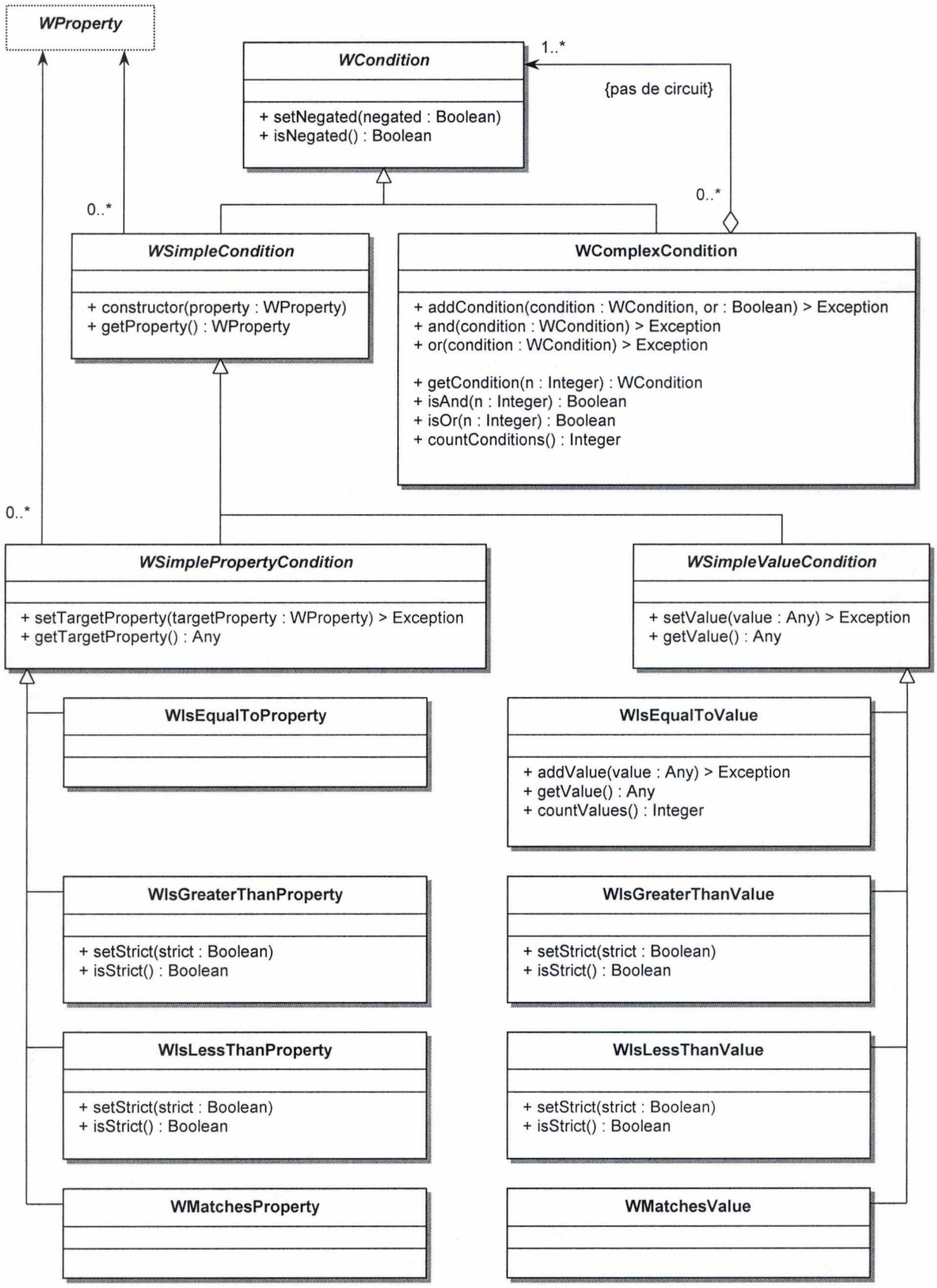

Figure IV-18 Diagramme des classes dédiées à la gestion des conditions. Tous les constructeurs ne sont pas présentés afin d'alléger le schéma.

Parcourons maintenant chacune des classes présentées et décrivons-en la modélisation.

a) WCondition

WCondition est la classe racine de toutes les autres classes gérant les conditions. Elle permet en outre d'inverser le sens d'une condition ("négativer").

Modélisation
+ setNegated(negated : Boolean)
Inverse la condition (équivalent de l'opérateur *not*) si le paramètre fourni est vrai.
+ isNegated() : Boolean
Indique si la condition est inversée.

b) WComplexCondition

WComplexCondition permet de construire une condition à partir d'une combinaison de plusieurs sous-conditions, connectées entre elles par les opérateurs de liaison *and* et *or*. Au sein d'une même instance de *WComplexCondition*, la priorité de ces deux opérateurs est d'application (*and* est prioritaire). Pour créer des conditions ne respectant pas cette priorité, plusieurs instances de *WComplexCondition* doivent intervenir.
Par exemple, pour créer la condition "A or B and C", c'est-à-dire "A or (B and C)", une seule instance de *WComplexCondition* suffit, comme illustré par la Figure IV-19 ci-dessous:

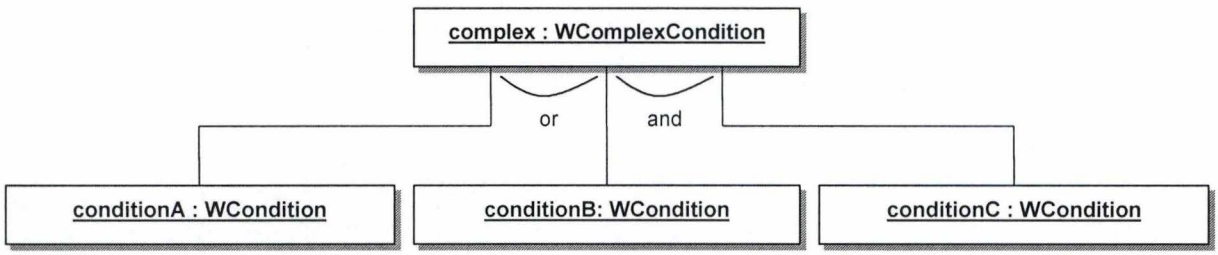

Figure IV-19 Diagramme d'instances illustrant la création de la condition "A or B and C". Une seule condition complexe est nécessaire puisque la condition à créer respecte la priorité des opérateurs de liaison.

Par contre, la création de la condition "(A or B) and C" requiert la participation de deux instances de *WComplexCondition*, comme l'illustre la Figure IV-20 ci-dessous:

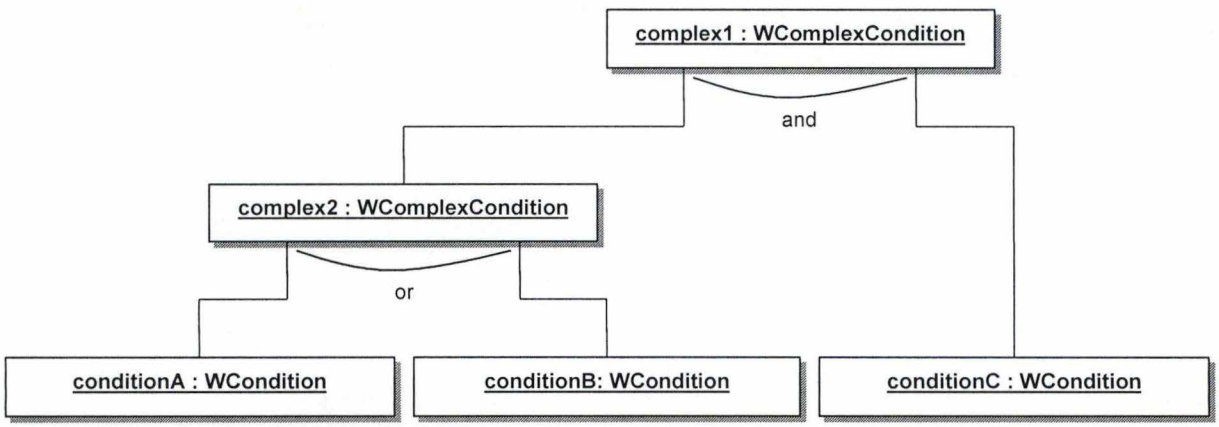

Figure IV-20 Diagramme d'instances illustrant la création de la condition "(A or B) and C". Deux conditions complexes sont nécessaires puisque la condition à créer ne respecte pas la priorité des opérateurs de liaison.

Insistons sur le fait que *WComplexCondition* autorise des combinaisons d'instances de toute classe dérivée de *WCondition*. Ainsi, dans les deux exemples ci-dessus, les conditions A, B et C peuvent représenter des instances de *WIsEqualToValue*, *WMatchesProperty*, *WComplexCondition*, ...

Notons également que l'exploitation d'une instance de *WComplexCondition* à laquelle aucune autre instance de *WCondition* n'est associée, sera sanctionnée par une exception.

Modélisation

+ addCondition(condition : WCondition, or : Boolean) > Exception
Associe une condition en la connectant à la dernière condition associée au moyen d'un opérateur de liaison: si le paramètre *or* est vrai, alors l'opérateur de liaison utilisé est *or*, sinon *and* est utilisé.
Déclenche une exception si l'instance de type *WCondition* fournie en paramètre est vide ou si elle est égale à l'instance courante: une condition complexe ne peut se référencer elle-même.
Notons que ce dernier test est un peu léger. Il requerrait en fait un système de contrôle plus complexe qui parcourerait toute l'arborescence de l'instance courante et de celle fournie en paramètre (car il peut très bien s'agir d'une condition complexe), et refuserait l'association si au moins une sous-condition de l'une se retrouvait dans l'autre (détection de circuits).

+ and(condition : WCondition) > Exception
Associe une condition en la liant à la dernière condition associée au moyen de l'opérateur *and* via la méthode *addCondition()*.

+ or(condition : WCondition) > Exception
Associe une condition en la liant à la dernière condition associée au moyen de l'opérateur *or* via la méthode *addCondition()*.

+ getCondition(n : Integer) : WCondition > Exception
Retourne la *n*ième condition associée ou déclenche une exception si *n* dépasse le nombre de conditions associées.

+ isAnd(n : Integer) : Boolean > Exception
Indique si la *n*ième condition est liée par l'opérateur de liaison *and* ou déclenche une exception si *n* dépasse le nombre de conditions associées.

+ isOr(n : Integer) : Boolean > Exception
Indique si la *n*ième condition est liée par l'opérateur de liaison *or* ou déclenche une exception si *n* dépasse le nombre de conditions associées.

+ countConditions() : Integer
Retourne le nombre de conditions associées.

c) WSimpleCondition

WSimpleCondition définit le minimum nécessaire pour la spécialisation de conditions simples, c'est-à-dire des conditions destinées à comparer la valeur de champ d'une instance de *WProperty* à "quelque chose d'autre".

Modélisation

+ constructor(property : WProperty) : WSimpleCondition > Exception
Crée une instance de *WSimpleCondition* et l'associe à une instance de *WProperty*. Une exception est déclenchée si l'instance de *WProperty* est vide.

+ getProperty() : WProperty
Retourne l'instance de *WProperty* associée.

d) WSimpleValueCondition et ses spécialisations

WSimpleValueCondition est la base nécessaire à la spécialisation de conditions destinées à comparer un champ encapsulé par une instance de *WProperty* à une valeur quelconque.

Modélisation

+ constructor(...) : WSimpleValueCondition > Exception
Crée une instance de *WSimpleValueCondition* et l'associe à une instance de *WProperty* (les ... remplacent les paramètres du constructeur de *WSimpleCondition*). La valeur de champ de l'instance de *WProperty* est utilisée pour la comparaison (via la méthode *getValue()* de *WProperty*). Une exception est déclenchée si l'instance de *WProperty* est vide.

+ constructor(property : WProperty, value : Any) : WSimpleValueCondition > Exception
Crée une instance de *WSimpleValueCondition* et l'associe à une instance de *WProperty* (via la méthode constructor de *WSimpleCondition*). La valeur passée en paramètre sera utilisée pour la comparaison, mais il est préalablement vérifié que son type est identique au type de données géré par l'instance de *WProperty*.
Une exception est déclenchée si l'instance de *WProperty* est vide ou si le type de la valeur fournie est incompatible avec le type de données gérées par l'instance de *WProperty*.

+ setValue(value : Any) > Exception
Assigne une valeur à laquelle le champ encapsulé par l'instance de *WProperty* fournie au constructeur devra être comparé. Déclenche une exception dans les mêmes circonstances qu'expliqué ci-dessus.

+ getValue() : Any
Retourne la valeur. Il s'agit soit de la valeur de l'instance de *WProperty* fournie au constructeur, soit d'une valeur fournie séparément, par exemple via *setValue()*.

WSimpleValueCondition dispose des spécialisations suivantes:

- **WIsEqualToValue.** Cette classe offre la possibilité de comparer un champ encapsulé par une instance de *WProperty* à une ou plusieurs valeurs. La comparaison à une seule valeur se traduit par un comparateur d'égalité (ou de différence si la condition est inversée), tandis que la comparaison à plusieurs valeurs se traduit par un comparateur d'inclusion (ou d'exclusion si la condition est inversée) à un ensemble. L'inclusion est l'équivalent de la clause SQL "IN (...)", et l'exclusion de la clause "NOT IN (...)".

Modélisation
+ constructor(..., negated : Boolean) : WIsEqualToValue > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimpleValueCondition*).
+ addValue(value : Any) > Exception
Ajoute une valeur supplémentaire à l'ensemble des valeurs auxquelles le champ encapsulé par l'instance de *WProperty* fournie au constructeur devra être comparé. Déclenche une exception dans les mêmes circonstances que pour la méthode *setValue()*.
Après le premier ajout, la cardinalité de l'ensemble de valeurs est 2 car la valeur assignable via la méthode *setValue()* est comprise dans le comptage.
+ getValue(n : Integer) : Any
Retourne la *n*ième valeur de l'ensemble. La première valeur est celle fournie via la méthode *setValue()* ou *constructor()*, la deuxième est la première valeur ajoutée via la méthode *addValue()*. Déclenche une exception si *n* dépasse le nombre de valeurs de l'ensemble.
+ countValues() : Integer
Retourne la cardinalité de l'ensemble de valeurs, c'est-à-dire le nombre de valeurs ajoutées via la méthode *addValue()* + 1.

- **WIsGreaterThanValue.** Cette classe permet de comparer un champ encapsulé par une instance de *WProperty* au moyen du comparateur "plus grand que" (ou "plus petit que ou égal à" si la condition est inversée), en spécifiant si la comparaison est stricte ou non.

Modélisation
+ constructor(..., strict : Boolean) : WIsGreaterThanValue > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimpleValueCondition*).
Spécifie en plus si la comparaison est stricte ou non stricte.
+ setStrict(strict : Boolean)
Spécifie si la comparaison est stricte ou non stricte.
+ isStrict(strict : Boolean)
Indique si la comparaison est stricte ou non stricte.

- **WIsLessThanValue.** Cette classe permet de comparer un champ encapsulé par une instance de *WProperty* au moyen du comparateur "plus petit que" (ou "plus grand que ou égal à" si la condition est inversée), en spécifiant si la comparaison est stricte ou non. Sa modélisation est équivalente à celle de la classe *WIsGreaterThanValue*.

Modélisation
+ constructor(..., strict : Boolean) : WIsLessThanValue > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimpleValueCondition*).
Spécifie en plus si la comparaison est stricte ou non stricte.
+ setStrict(strict : Boolean)
Spécifie si la comparaison est stricte ou non stricte.
+ isStrict(strict : Boolean)
Indique si la comparaison est stricte ou non stricte.

- **WMatchesValue.** *WMatchesValue* permet de comparer un champ encapsulé par une instance de *WProperty* au moyen d'un comparateur de type "pattern matching", correspondant à la clause SQL "LIKE".

Modélisation
+ constructor(...) : WMatchesValue > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimpleValueCondition*).

e) WSimplePropertyCondition et ses spécialisations

WSimplePropertyCondition définit la classe servant de base aux spécialisations de conditions dédiées à la comparaison d'un champ encapsulé par une instance de *WProperty* à un autre champ encapsulé par une autre instance de *WProperty*. Ces deux champs peuvent appartenir à deux tables distinctes, dans quel cas la condition aura l'effet d'un "join".

Modélisation
+ constructor(..., targetProperty : WProperty) : WSimplePropertyCondition > Exception
Crée une instance de *WSimplePropertyCondition* (les ... remplacent les paramètres du constructeur de *WSimpleCondition*). On associe en plus la deuxième instance de *WProperty* (*targetProperty*) qui encapsule le champ auquel il faudra comparer le champ encapsulé par la première instance de *WProperty* (paramètre *property*). Déclenche une exception si au moins une des deux instances de *WProperty* est vide.
+ setTargetProperty(targetProperty : WProperty) > Exception
Assigne une instance de *WProperty* encapsulant le champ auquel il faudra comparer le champ encapsulé par l'instance de *WProperty* fournie au constructeur. Déclenche une exception si cette instance de *WProperty* est vide.
+ getTargetProperty() : WProperty
Retourne l'instance de *WProperty* fournie via la méthode *setTargetProperty()* ou *constructor()*.

On remarque que nous avons ici d'office deux paramètres au constructeur. La raison est que les deux instances de *WProperty* fournie au constructeur sont indispensables au fonctionnement de la classe, contrairement à *WSimpleValueCondition*.

WSimplePropertyCondition dispose des spécialisations suivantes:

- **WIsEqualToProperty.** Cette classe gère le comparateur d'égalité (ou de différence si la condition est inversée) entre deux champs.

Modélisation
+ constructor(...) : WIsEqualToProperty > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimplePropertyCondition*).

- **WGreaterThanProperty.** Cette classe gère le comparateur "plus grand que" (ou "plus petit que ou égal à" si la condition est inversée) entre deux champs.

Modélisation
+ constructor(..., strict : Boolean) : WGreaterThanProperty > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimplePropertyCondition*). Spécifie en plus si la comparaison est stricte ou non stricte.
+ setStrict(strict : Boolean)
Spécifie si la comparaison est stricte ou non stricte.
+ isStrict(strict : Boolean)
Indique si la comparaison est stricte ou non stricte.

- **WLessThanProperty.** Cette classe gère le comparateur "plus petit que" (ou "plus grand que ou égal à" si la condition est inversée) entre deux champs.

Modélisation
+ constructor(..., strict : Boolean) : WLessThanProperty> Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimplePropertyCondition*). Spécifie en plus si la comparaison est stricte ou non stricte.
+ setStrict(strict : Boolean)
Spécifie si la comparaison est stricte ou non stricte.
+ isStrict(strict : Boolean)
Indique si la comparaison est stricte ou non stricte.

- **WMatchesProperty.** Cette classe gère le comparateur de "pattern matching", correspondant à la clause SQL "LIKE", entre deux champs.

Modélisation
+ constructor(...) : WMatchesProperty > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WSimplePropertyCondition*).

f) Exemples

La figure ci-dessous fournit des exemples d'utilisation de *WCondition* en Java:

```
// Déclaration des instances que nous allons utiliser.
XClient   client   = new XClient();
XCommande commande = new XCommande();

// Condition COMMANDE.ID = "123"
new WIsEqualToValue(commande.id, "123");

// Condition COMMANDE.VERSION > "10" refusée (une exception est déclenchée) car le type
// de commande.version n'est pas String mais bien Integer.
new WIsGreaterThanValue(commande.version, "10");

// Condition COMMANDE.CLIENT_ID = CLIENT.ID
new WIsEqualToValue(commande.clientId, client.id);

// Condition ClIENT.NOM dans {"Tintin", "Haddock", "Tournesol"}
WIsEqualToValue c1 = new WIsEqualToValue(client.nom);
c1.setValue("Tintin");
c1.addValue("Haddock");
c1.addValue("Tournesol");

// ... ou bien ...
WComplexCondition c2 = new WComplexCondition();
c2.or(new WIsEqualToValue(client.nom, "Tintin"));
c2.or(new WIsEqualToValue(client.nom, "Haddock"));
c2.or(new WIsEqualToValue(client.nom, "Tournesol"));

// Condition CLIENT.NOM <= "Dupont"
new WIsLessThanValue(client.nom, "Dupont", false);

// ... ou bien ...
WIsGreaterThanValue c3 = new WIsGreaterThanValue(client.nom, "Dupont", true);
c3.setNegated(true);

// Condition (COMMANDE.DATE_COMMANDE = COMMANDE.DATE_LIVRAISON) et
//           (CLIENT.NOM = "Baxter" ou CLIENT.NOM = "Wolf")
WComplexCondition c4 = new WComplexCondition();
WComplexCondition c5 = new WComplexCondition();
c4.or(new WIsEqualToValue(client.nom, "Baxter"));
c4.or(new WIsEqualToValue(client.nom, "Wolf"));
c5.and(new WIsEqualToProperty(commande.dateCommande, commande.dateLivraison);
c5.and(c4);
```

Figure IV-21 Exemples d'utilisation de *WCondition* en Java

IV.2.2 WSortOrder

WSortOrder fournit les structures nécessaires à la spécification de l'ordre dans lequel les enregistrements sont chargés. L'ordre est défini par une suite d'instances de *WProperty*, chacune d'entre elle étant assortie d'une direction (ascendante ou descendante).

Modélisation
+ addProperty(property : WProperty, ascending : Boolean) > Exception
Ajoute une instance de *WProperty* dans la suite des instances déterminant l'ordre de tri, en spécifiant si l'ordre est croissant ou décroissant. Déclenche une exception si l'instance de *WProperty* est vide.
+ ascending(property : WProperty) > Exception
Idem que pour la méthode *addProperty()* mais en forçant un ordre de tri croissant

+ descending(property : WProperty) > Exception
Idem que pour la méthode *addProperty()* mais en forçant un ordre de tri décroissant
+ getProperty(n : Integer) : WProperty > Exception
Retourne la *n*ième instance de *WProperty* associée ou déclenche une exception si *n*dépasse le nombre d'instances de *WProperty* associées.
+ isAscending(n : Integer) : Boolean > Exception
Indique si la *n*ième instance de *WProperty* associée spécifie un ordre croissant ou déclenche une exception si *n* dépasse le nombre d'instances de *WProperty* associées.
+ countProperties() : Integer
Retourne le nombre d'instances de *WProperty* associées.

Des exemples sont proposés plus loin, conjointement à ceux de *WCollection*.

IV.2.3 WCollection

Après avoir vu les conditions et les critères de tri, nous sommes en possession de suffisamment d'éléments pour aborder la description de la classe *WCollection*. Remarque intéressante: *WCollection*, contrairement à *WObject*, n'est pas une classe abstraite et peut donc être exploitée telle quelle.

a) Association à WObject

WCollection permet de sélectionner des enregistrements issus d'une ou plusieurs tables. L'association à des instances de *WObject* détermine de quelles tables (ou vues) les enregistrements doivent être extraits (une même instance de *WObject* étant connectée à une et une seule table). Il est permis d'associer plusieurs instances de *WObject* connectées à la même table, pour gérer les références cycliques.
De plus, la désignation d'une jointure entre deux instances de *WObject* est possible en spécifiant une instance de *WReference* permettant de naviguer de l'instance de *WObject* source vers l'instance de *WObject* cible. Les jointures peuvent également être spécifiées sous la forme de conditions (cfr ci-après, Critères de sélection)

Modélisation
+ registerObject(object : WObject) > Exception
Associe une instance de *WObject*. Déclenche une exception si l'instance de *WObject* est vide.
+ registerObject(object : WObject, joinReference : WReference) > Exception
Associe une instance de *WObject* et établit une jointure entre cette instance et l'instance de *WObject* à laquelle est associée une instance de *WReference*. Déclenche une exception si l'instance de *WObject* est vide.
+ getObject(n : Integer) : WObject > Exception
Retourne la *n*ième instance de *WObject* associée ou déclenche une exception si *n* dépasse le nombre d'instances de *WObject* associées.
+ countObjects() : Integer
Retourne le nombre d'instances de *WObject* associées.

b) Critères de sélection

L'association à une instance de *WCondition* permet de filtrer les enregistrements à rapatrier et de spécifier les jointures entre tables.

Modélisation
+ setCondition(condition : WCondition) > Exception
Associe une instance de *WCondition*. Déclenche une exception si elle est vide.
+ getCondition() : WCondition
Retourne l'instance de *WCondition* préalablement associée.

c) Ordre de tri

L'association à une instance de *WSortOrder* permet de sélectionner les enregistrements suivant un ordre donné.

Modélisation

+ setSortOrder(sortOrder : WSortOrder) > Exception
Associe une instance de *WSortOrder*. Déclenche une exception si elle est vide.

+ getSortOrder() : WSortOrder
Retourne l'instance de *WSortOrder* préalablement associée.

d) Opérations

La principale utilité de *WCollection* est le parcours des enregistrements. A chaque itération du parcours, les instances de *WObject* associées sont automatiquement chargées avec les valeurs de champs de l'enregistrement courant. Le nombre total d'enregistrements à rapatrier peut être limité et il est possible de compter le nombre total d'enregistrements répondant à la condition courante (l'instance de *WCondition* associée) indépendamment de cette limite.
Le parcours peut être arrêté et reinitialisé à tout moment. L'association d'une nouvelle instance de *WObject*, de *WCondition* ou de *WSortOrder*, ainsi que le changement du nombre total d'enregistrements à rapatrier réinitialisent automatiquement le parcours. La prochaine itération se positionnera donc sur le premier enregistrement (s'il existe).

Deux opérations de mise à jour sont également prévues. L'opération de modification met à jour tous les enregistrements encapsulés par une instance de *WObject* donnée, suivant les critères de conditions en cours, avec les valeurs de champs des instances de *WProperty* dont l'état est "altéré".
L'opération de suppression supprime tous les enregistrements encapsulés par une instance de *WObject* donnée, suivant les critères de conditions en cours.

Modélisation

+ refresh()
Réinitialise le parcours en cours! Cette méthode est appelée d'office par les méthodes *registerObject()*, *setCondition()*, *setSortOrder()* et *setMaxNexts()*.

+ next() : Boolean > Exception
Charge l'enregistrement suivant dans les instances de *WObject* associées. Lors du premier appel à cette méthode après une réinitialisation, le premier enregistrement est chargé s'il existe. Retourne vrai si un enregistrement a été chargé. Sinon, retourne faux et réinitialise toutes les instances de *WObject* associées (via leur méthode *reset()*). Déclenche une exception si cette méthode est appelée alors que l'appel précédent avait déjà retourné faux. En termes d'états, cette méthode peut être appelée tant que la méthode *isAfterLast()* retourne faux (autrement dit tant que la méthode *next()* retourne vrai).
Une exception est également déclenchée si aucune instance de *WObject* n'est associée.

+ count() > Exception
Retourne le nombre d'enregistrements répondant à l'instance de *WCondition* courante, ou le nombre total d'enregistrements si aucune instance de *WCondition* n'est associée.

+ setMaxNexts(maxNexts : Integer)
Limite le nombre d'enregistrements à rapatrier de la base de données. Lors d'un parcours, la méthode *next()* retournera faux au plus tard lorsqu'elle aura dépassé cette limite.

+ getMaxNexts() : Integer
Retourne la limite fixée via la méthode *setMaxNexts()*.

+ modify(object : WObject) > Exception
Modifie tous les enregistrements encapsulés par l'instance de *WObject* spécifiée et répondants à l'instance de *WCondition* courante, avec les valeurs de champs des instances de *WProperty* (associées à ladite instance de *WObject*) dont l'état est "altéré" (cfr méthode *isDirty()* de *WProperty*).
Déclenche une exception si l'instance de *WObject* spécifiée est vide, si cette instance n'est pas associée à l'instance de *WCollection* courante ou si aucune des instances de *WProperty* associées n'a un état égal à "altéré".

+ remove(object : WObject) > Exception
Supprime tous les enregistrements encapsulés par l'instance de *WObject* spécifiée et répondants à l'instance de *WCondition* courante.
Déclenche une exception si l'instance de *WObject* spécifiée est vide ou si cette instance n'est pas associée à l'instance de *WCollection* courante.

e) Etats

L'état d'une instance de *WCollection* se résume à définir la position de l'enregistrement courant parmi l'ensemble à parcourir et ce, sous une forme soit absolue (le numéro de l'enregistrement courant), soit relative (la situation par rapport au premier et au dernier enregistrement).

Modélisation

+ getPosition() : Integer
Retourne le numéro de l'enregistrement courant, ou -1 si aucun enregistrement n'est disponible.

+ isBeforeFirst() : Boolean
Indique si le premier enregistrement n'a pas encore été chargé. Une instance de *WCollection* se trouve dans cet état après l'appel à la méthode *refresh()* et avant l'appel à la méthode *next()*.

+ isFirst() : Boolean
Indique si l'instance de *WCollection* courante est positionnée sur le premier enregistrement, c'est-à-dire l'enregistrement chargé via le premier appel à la méthode *next()* (ayant retourné vrai).

+ isLast() : Boolean
Indique si l'instance de *WCollection* courante est positionnée sur le dernier enregistrement disponible.

+ isAfterLast() : Boolean
Indique si l'instance de *WCollection* courante est positionnée après le dernier enregistrement disponible. Autrement dit, le dernier appel à la méthode *next()* a retourné faux.

f) Modélisation

La Figure IV-22 présente une vue générale de la modélisation de *WCollection*:

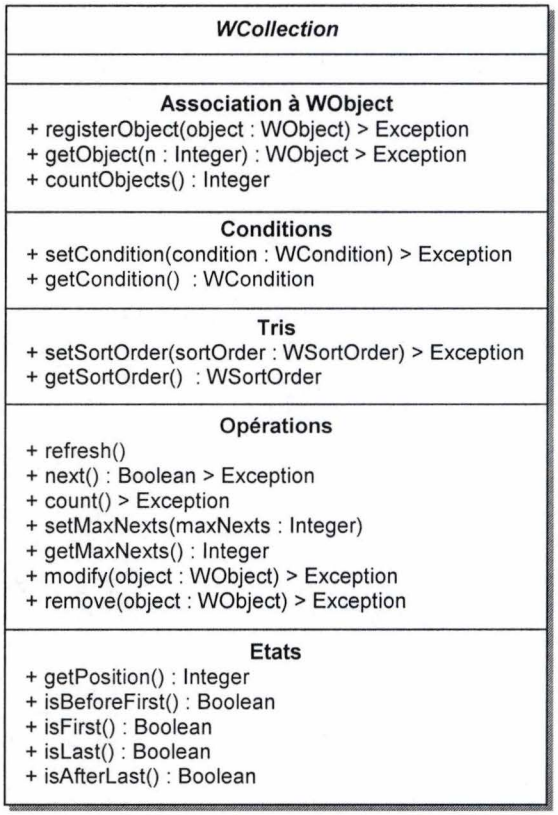

Figure IV-22 Modélisation de *WCollection*

g) Exemples

Nous proposons deux séries d'exemples. La première série, illustrée par la Figure IV-23, exploite *WCollection* telle quelle, la deuxième utilise une spécialisation de cette classe et est présentée plus loin dans la Figure IV-24.

```
// Déclaration des instances de WObject que nous allons utiliser.
XClient   client   = new XClient();
XCommande commande = new XCommande();
XDétail   détail   = new XDétail();

// Création d'une instance de WCondition.
WComplexCondition cond = new WComplexCondition();
cond.and(new WIsEqualToValue(détail.quantité, new Integer(1)));
cond.and(new WIsLessThanValue(détail.prix, new Integer(100)));
```

```
// Les résultats seront triés par la date de commande et le nom du client
WSortOrder sort = new WSortOrder();
sort.ascending(commande.dateCommande);
sort.ascending(client.nom);

// Création d'une instance de WCollection et association aux instances de WObject,
// WCondition et WSortOrder.
//
// L'équivalent en SQL est:
//   SELECT CLIENT.*, COMMANDE.*, DETAIL.*
//   FROM CLIENT, COMMANDE, DETAIL
//   WHERE COMMANDE.CLIENT_ID = CLIENT.ID
//     AND DETAIL.COMMANDE_ID = COMMANDE.ID
//     AND DETAIL.QUANTITE = 1
//     AND DETAIL.PRIX <= 100
WCollection coll = new WCollection();
coll.registerObject(détail);
coll.registerObject(commande, détail.commandeRef); // Jointure DETAIL->COMMANDE
coll.registerObject(client, commande.clientRef);   // Jointure COMMANDE->CLIENT
coll.setCondition(cond);
coll.setSortOrder(sort);

System.out.println(coll.isBeforeFirst()); // Affiche 'true'
System.out.println(coll.isAfterLast());   // Affiche 'false'

// Parcours et affichage des résultats
while(coll.next())
    System.out.println("" + coll.getPosition() + " " +
                         client.nom + " " +
                         detail.prix + " " +
                         commande.dateCommande);

System.out.println(coll.isBeforeFirst()); // Affiche 'false'
System.out.println(coll.isAfterLast());   // Affiche 'true'
```

Figure IV-23 Exemple d'utilisation de WCollection tel quel.

Voyons maintenant un exemple de spécialisation dans la Figure IV-24:

```
// On crée une petite classe héritée de WCollection et qui encapsule un objet de type
// XClient
class XClients extends WCollection
{
  public XClient client = null;

  public XClients()
  {
    client = new XClient();
    registerObject(client);
  }
}

// On instancie cette classe
XClients clients = new XClients();

// On compte le nombre de clients
System.out.println(clients.count());

// On parcourt tous les clients
while(clients.next())
  System.out.println(clients.client.nom);
```

Figure IV-24 Exemple de spécialisation de *WCollection*.

IV.3 Transactions

Nous avons déjà évoqué à plusieurs reprises que le rôle des transactions du modèle est d'offrir un mécanisme de synchronisation entre les transactions de la base de données et les instances de classes.

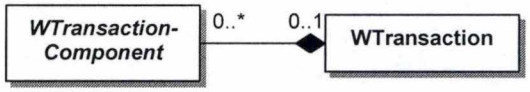

Figure IV-25 Diagramme de classe des composants intervenant dans la gestion des transactions

La Figure IV-25 illustre les deux composants jouant un rôle dans la gestion des transactions: *WTransactionComponent* et *WTransation*.

IV.3.1 WTransactionComponent

WTransactionComponent est une classe entièrement abstraite définissant un ensemble de méthodes que les classes désirant bénéficier des services transactionnels sont tenues d'implémenter, comme le font les classes *WObject* et *WCollection* de notre modèle. Ces dernières sont d'ailleurs représentées sur la Figure IV-26.

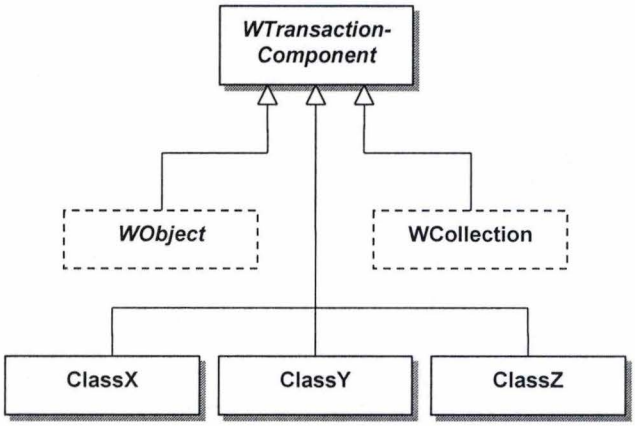

Figure IV-26 Les classes désirant participer aux transactions doivent spécialiser *WTransactionComponent*. *WObject* et *WCollection* en font partie.

On remarque que d'autres classes a priori étrangères à notre modèle (les classes *ClassX, Y* et *Z*) peuvent également spécialiser la classe *WTransactionComponent* de laquelle elles doivent alors implémenter toutes les méthodes abstraites. C'est la condition sine qua non pour pouvoir participer à une transaction, c'est-à-dire s'associer à une instance de *WTransaction*. En fait, ces méthodes abstraites n'ont de sens que parce qu'elles sont utilisées par *WTransaction*.

a) Participation à une transaction

Un événement est déclenché au moment où une instance de classe héritée de *WTransactionComponent* est soit associée à une instance de *WTransaction*, soit éjectée d'une instance de *WTransaction* (lorsque celle-ci est "nettoyée"). De plus cette instance doit être capable de fournir l'instance de *WTransaction* à laquelle elle est associée.

Modélisation

 # joiningTransaction(transaction : WTransaction)

Informe l'instance courante qu'elle vient d'être associée à une instance de *WTransaction*. L'instance courante doit conserver l'instance de *WTransaction* fournie en paramètre pour pouvoir la restituer plus tard (via la méthode *getTransaction()*).
Cette méthode est protégée car elle ne peut être appelée que par l'instance de *WTransaction* associée (via sa méthode *registerComponent()*).

 # leavingTransaction()

Informe l'instance courante qu'elle vient d'être éjectée de l'instance de *WTransaction* à laquelle elle était associée.
Cette méthode est protégée car elle ne peut être appelée que par l'instance de *WTransaction* associée (via sa méthode *cleanup()*).

+ getTransaction() : WTransaction

Si l'instance courante est associée à une instance de *WTransaction*, retourne cette instance, sinon retourne "vide".

b) Validation et annulation d'une transaction

Lors de la validation ou de l'annulation d'une transaction, tous les composants associés à cette transaction en sont avertis par l'intermédiraire d'événements. Un premier événement est déclenché avant la validation/annulation proprement dite, et un deuxième après la complétion de la validation/annulation.
Si pour une raison quelconque, un composant est dans l'incapacité de participer à la validation/annulation, il doit déclencher une exception en réponse au premier événement. En conséquence, la validation/annulation de la transaction ne sera pas démarrée.
Le deuxième événement est déclenché après la validation/annulation de la transaction au niveau de la base de données. Typiquement, en réponse à cet événement, une instance de *WTransactionComponent* va soit effectuer une sauvegarde de son état (dans le cas d'une validation), soit restaurer son état (dans le cas d'une annulation).

Modélisation

 # willCompleteTransaction() > Exception

Informe l'instance courante que l'instance de *WTransaction* associée va démarrer une opération de validation. L'instance courante peut déclencher une exception si, à ce moment, elle se sait incapable d'assumer une validation. Cette exception empêchera le démarrage effectif de la validation de l'instance de *WTransaction* associée.
Cette méthode est protégée car elle ne peut être appelée que par l'instance de *WTransaction* associée (via sa méthode *complete()*).

 # completingTransaction()

Informe l'instance courante que l'instance de *WTransaction* associée a effectivement démarré l'opération de validation et que les modifications effectuées sur la base de données ont déjà été validées.
Cette méthode est protégée car elle ne peut être appelée que par l'instance de *WTransaction* associée (via sa méthode *complete()*).

 # willAbortTransaction() > Exception

Informe l'instance courante que l'instance de *WTransaction* associée va démarrer une opération d'annulation. L'instance courante peut déclencher une exception si, à ce moment, elle se sait incapable d'assumer une annulation. Cette exception empêchera le démarrage effectif de l'annulation de l'instance de *WTransaction* associée.
Cette méthode est protégée car elle ne peut être appelée que par l'instance de *WTransaction* associée (via sa méthode *abort()*).

 # abortingTransaction()

Informe l'instance courante que l'instance de *WTransaction* associée a effectivement démarré l'opération d'annulation et que les modifications effectuées sur la base de données ont déjà été annulées.
Cette méthode est protégée car elle ne peut être appelée que par l'instance de *WTransaction* associée (via sa méthode *abort()*).

c) Modélisation

La Figure IV-27 présente la modélisation de *WTransactionComponent*.

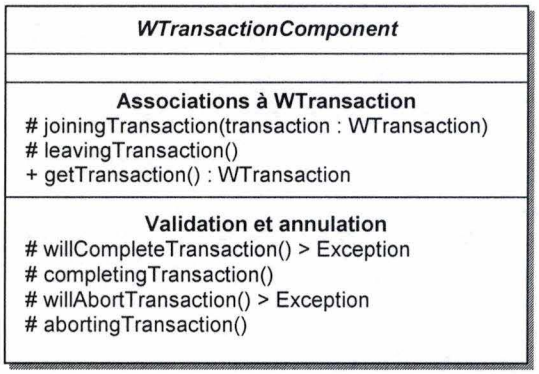

Figure IV-27 Modélisation de *WTransactionComponent*

d) Exemples

Les exemples relatifs à *WTransactionComponent* sont présentés conjointement à ceux de *WTransaction*.

IV.3.2 WTransaction

WTransaction prend en charge la gestion des transactions proprement dites. Il s'agit d'une classe concrète (donc directement exploitable) qui permet de synchroniser les transactions au niveau des instances de *WTransactionComponent* associées avec les transactions de la base de données sous-jacente.

Les classes héritées de *WTransactionComponent* doivent définir comment elles réagissent dans un cadre transactionnel. Les classes *WObject* et *WCollection* de notre modèle se comportent comme suit:

- **WObject**. Lors du démarrage d'une transaction (ou lors de l'association d'une instance de *WObject* à une transaction), les instances de *WObject* associées vont sauvegarder leur l'état et déclencher une opération de sauvegarde sur les instances de *WProperty* associées. Par défaut, seules les données dynamiques sont sauvegardées, et donc pas les données statiques comme les étiquettes et les descriptions. Il est toutefois possible de spécialiser *WObject* pour étendre la sauvegarde à d'autres données.

  Lors d'une annulation, les données sauvegardées au démarrage de la transaction sont restituées. Ainsi, chaque instance de *WObject* se retrouve dans une configuration opérationnelle équivalente à celle qu'elle avait au démarrage de la transaction.

- **WCollection**. Lors de l'annulation d'une transaction (via l'instance de *WTransaction* associée), une instance de *WCollection* doit pouvoir se repositionner sur l'enregistrement qui était courant lors du démarrage de cette transaction et restituer l'état d'origine de ladite instance de *WCollection*. Ce mécanisme relève de l'implémentation et n'est donc pas détaillé ici.

a) Association à WTransactionComponent

Un nombre indéfini d'instances de *WTransactionComponent* peuvent être associées à une même instance de *WTransaction*. Il est impossible de désassocier une instance de *WTransactionComponent* de l'instance de *WTransaction* à laquelle elle est associée. Pour ce faire, la transaction dans son entièreté doit être nettoyée, comme nous l'expliquons plus loin (cfr ci-après, Opérations).

Modélisation

+ registerComponent(component : WTransactionComponent) > Exception
Associe une instance de *WTransactionComponent*. Déclenche une exception si cette instance de *WTransactionComponent* est vide ou si elle est déjà associée à une autre instance de *WTransaction*.

+ containsComponent(component : WTransactionComponent) : Boolean
Indique si une instance de *WTransactionComponent* est associée à l'instance courante de *WTransaction*.

+ getComponent(n : Integer) : WTransactionComponent > Exception
Retourne la *n*ième instance de *WTransactionComponent* associée ou déclenche une exception si *n* dépasse le nombre d'instances de *WTransactionComponent* associées.

+ countComponents() : Integer
Retourne le nombre d'instances de *WTransactionComponent* associées.

b) Opérations

Les opérations possibles sont au nombre de quatre: démarrage, validation, annulation et nettoyage. Le démarrage est implicite et fait suite à une validation, une annulation ou la création d'une instance de *WTransaction*.

Lors d'une validation/annulation, les instances de *WTransactionComponent* associées sont préalablement averties par l'intermédiaire d'événements (cfr plus haut, IV.3.1). Ensuite, le cas échéant, une validation/annulation est exécutée sur la base de données. Enfin, les instances de *WTransactionComponent* associées sont informées (également par le biais d'un événement) que la validation/annulation se termine. La prise en charge de la sauvegarde et de la restitution (si la transaction est annulée) de l'état d'une instance de *WTransactionComponent* est de la compétence de cette instance.

Par exemple, lors d'une annulation, une instance de *WObject* va rétablir l'état qu'elle avait au démarrage de cette transaction, c'est-à-dire qu'elle va restaurer son propre état, ainsi que les valeurs de champs et les états de ses instances de *WProperty* associées. Cette procédure est gérée par la classe *WObject* elle-même en réponse aux événements déclenchés par *WTransaction*.

Le nettoyage d'une transaction dissocie toutes les instances de *WTransactionComponent* associées et envoie un ordre d'annulation à la base de données.

Modélisation

+ complete() > Exception
Valide la transaction en trois étapes:
- informe les instances de *WTransactionComponent* associées qu'une validation va débuter (cfr l'événement *willCompleteTransaction()* de *WTransactionComponent*)
- valide les éventuelles modifications effectuées sur la base de données
- informe les instances de *WTransactionComponent* que la validation est en phase terminale (cfr l'événement *completingTransaction()* de *WTransactionComponent*)

Une exception est déclenchée si une des instances de *WTransactionComponent* associées a elle-même déclenché une exception lors de la première étape, ou si la validation de la deuxième étape échoue.

+ abort() > Exception
Annule la transaction en trois étapes:
- informe les instances de *WTransactionComponent* associées qu'une annulation va débuter (cfr l'événement *willAbortTransaction()* de *WTransactionComponent*)
- annule les éventuelles modifications effectuées sur la base de données
- informe les instances de *WTransactionComponent* que l'annulation est en phase terminale (cfr l'événement *abortingTransaction()* de *WTransactionComponent*)

Une exception est déclenchée si une des instances de *WTransactionComponent* associées a elle-même déclenché une exception lors de la première étape, ou si l'annulation de la deuxième étape échoue.

+ cleanup() > Exception
Nettoie la transaction en trois étapes:
- annule les éventuelles modifications effectuées sur la base de données
- dissocie les instances de *WTransactionComponent* associées
- informe ces instances qu'elles on été éjectées de l'instance de *WTransaction* courante (cfr l'événement *leavingTransaction()* de *WTransactionComponent*)

c) Etats

L'exécution d'une opération (réussie ou non) donne lieu à un changement d'état. On distingue les états suivants:

- "nouveau" si l'instance de *WTransaction* vient d'être créée
- "validation en cours" si une opération de validation va débuter
- "validation terminée" si une opération de validation est terminée
- "annulation en cours" si une opération d'annulation va débuter
- "annulation terminée" si une opération d'annulation est terminée
- "nettoyage en cours" si une opération de nettoyage va débuter
- "nettoyage terminé" si une opération de nettoyage est terminée

Modélisation
+ getStatus() : WTransactionStatusEnumeration
Retourne l'état de l'instance courante.
Le type WObject*StatusEnumeration* contient les valeurs {*new*, *validating*, *validated*, *aborting*, *aborted*, *cleaningup*, *cleanup*}.

d) Modélisation

La Figure IV-27 présente la modélisation de *WTransaction*.

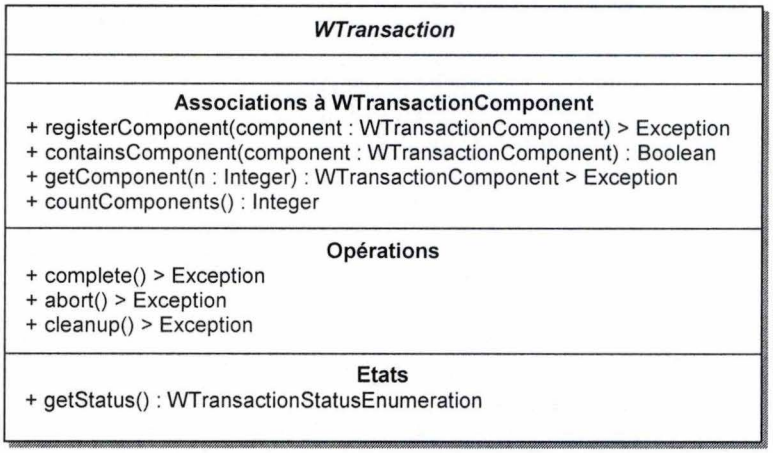

Figure IV-28 Modélisation de *WTransaction*

e) Exemples

Le premier exemple présenté par la Figure IV-29 illustre le comportement des *WObject* lorsqu'ils participent à une transaction, le deuxième exemple examinera les *WCollection* (cfr Figure IV-30).

```
// Déclaration de l'instance de WObject que nous allons utiliser.
XClient client = new XClient();

// Déclaration d'un objet WTransaction
WTransaction tran = new WTransaction();

// Associe l'objet client à la transaction
tran.registerComponent(client);

// Charge le client dont le nom est "Boss" (on suppose qu'il existe), remplace son
// nom par "Rastapopoulos", modifie l'enregistrement et valide la transaction.
client.load(new WIsEqualToValue(client.nom, "Boss"));
client.nom.setValue("Rastapopoulos");
client.modify();
tran.complete();
```

```
// Remplace le nom du client (càd "Rastapopoulos") par "Alan", modifie
// l'enregistrement, mais annule la transaction.
// L'objet client se retrouve donc dans l'état dans lequel il était juste après l'appel
// à la méthode complete() ci-dessus, et les modifications sur la base de données
// sont annulées.
client.nom.setValue("Alan");
client.modify();
tran.abort();

// Affiche le nom du client càd "Rastapopoulos"
System.out.println(client.nom);
```

Figure IV-29 Exemple de *WObject* dans une transaction

```
// Déclaration de l'instance de WObject que nous allons utiliser.
XClient client = new XClient();

// Déclaration d'un objet WTransaction
WTransaction tran = new WTransaction();

// Déclaration d'un objet WCollection
WCollection col = new WCollection();

// Association de l'object client à la collection
col.registerObject(client);

// Association de la collection et de l'objet client à la transaction.
tran.registerComponent(col);
tran.registerComponent(client);

// Se positionne sur le deuxième enregistrement (on suppose qu'il existe) et
// on valide la transaction
col.next();
col.next();
tran.complete();

// On lit le troisième enregistrement (on suppose qu'il existe) et
// on annule la transaction. L'enregistrement courant redevient le deuxième,
// et l'objet client récupère le contenu du deuxième enregistrement puisqu'il
// participe également à la transaction.
col.next();
tran.abort();
```

Figure IV-30 Exemple de *WCollection* dans une transaction

IV.4 Limites et extensions potentielles

Le langage que nous venons de définir répond aux spécifications décrites dans le chapitre précédent. Il est donc naturel que les limites qui avaient alors été relevées, soient ici implicitement d'application. Nous n'allons donc plus les évoquer mais nous concentrer sur les limites apparues au cours de ce chapitre.

Chacune de ces limites présente l'intérêt d'ouvrir la porte à de nouvelles fonctionnalités qu'il serait intéressant d'implémenter.

- **Cardinalités.** Dans *WProperty*, seules les cardinalités [0-1] et [1-1] sont gérées, c'est-à-dire les cardinalités exprimant le caractère requis ou non-requis du champ.

- **Formules.** Les formules de mise à jour de champs impliquant d'autres champs ne sont pas permises. Par exemple, il est impossible de majorer de 2% le prix de tous les produits en une seule requête car l'instruction devrait se formuler comme suit: prix = prix * 1.02. La solution est de d'abord charger l'enregistrement, de lui appliquer la formule et ensuite de le sauvegarder.

  On observe le même type de problème pour les conditions. Il est en effet impossible de créer une condition du type "*PRODUIT.PRIX* < *DETAIL.PRIX* * 0.9", par contre "*PRODUIT.PRIX* < *DETAIL.PRIX*" est possible.

- **Agrégations.** La sémantique des agrégations telle que proposée par SQL (MIN, MAX, GROUP BY, HAVING, ...) n'a aucun équivalent direct dans notre langage. Il faut dès lors simuler ces fonctionnalités en parcourant les enregistrements un par un.

- **Requêtes imbriquées.** Pour l'instant, aucune sémantique n'est prévue pour exprimer la requête suivante: la liste des clients ayant passé au moins une commande après une date *x*.

```
SELECT *
FROM CLIENT
WHERE ID IN (
  SELECT DISTINCT CLIENT_ID
  FROM COMMANDE
  WHERE DATE_COMMANDE > x)
```

  On pourrait pallier ce problème en élargissant les fonctionnalités de la classe *WIsEqualToProperty* qui implémenterait le nouveau constructeur suivant:

```
constructor(property : WProperty, targetProperty : WProperty, condition : WCondition) : WIsEqualToProperty
```

  En Java, la condition de la requête SQL ci-dessus s'exprimerait alors comme suit:

```
new WIsEqualToProperty(client.id, commande.clientId, new WIsGreaterThanValue(commande.dateCommande, x))
```

- **Définition des objets.** Avant de pouvoir manipuler la base de données sous-jacente, il est indispensable de définir un ensemble de *WObject* et *WProperty*. Toutefois, cette tâche est non récurrente puisque, une fois définis, ces objets sont réutilisables à souhait.

  Comme nous l'avons déjà signalé, la présence d'un générateur de classes travaillant à partir de schémas de bases de données serait idéal.

  La construction des objets est examinée dans le chapitre suivant.

IV.5 Exemples applicatifs

Tout d'abord un pot pourri d'exemples offre un aperçu des fonctionnalités du langage que nous venons d'étudier. Ensuite, nous proposons un morceau de programme dédié à l'encodage d'une commande.

IV.5.1 Exemples divers

```
// Déclaration des instances des classes encapsulant les tables de la base de données
// illustrative. Ces classes ont été définies plus haut conjointement à la description
// du noyau (cfr IV.1.3, Modélisation de la base de données illustrative).
XClient   client   = new XClient();
XCommande commande = new XCommande();
XDétail   détail   = new XDétail();
XProduit  produit  = new XProduit();
XProduit  produit2 = new XProduit();

// Chargement d'une commande à partir d'une valeur d'identifiant et affichage de la
// date du passage de la commande.
commande.id.setValue("AAAAAAAAAAAAAAAA");
commande.load();
System.out.println("Date: " + commande.dateCommande);

// Même exemple que ci-dessus, mais en utilisant une instance de WCondition
commande.load(new WIsEqualToValue(commande.id, "AAAAAAAAAAAAAAAA"));
System.out.println("Date: " + commande.dateCommande);

// Chargement d'un client à partir de son code en utilisant une instance de WCondition.
// Même si client.code ne participe pas à l'identification de l'enregistrement au sein
// de client, nous savons que CODE est unique pour la table CLIENT, puisqu'il est
// décrit comme tel dans le diagramme de notre base de données illustrative.
// L'opération de chargement va donc retourner un enregistrement au plus, ce qui est
// parfaitement accepté par WObject.
client.load(new WIsEqualToValue(client.code, "CASTAFIORE"));
System.out.println("Client: " + client.nom);

// Navigation simple vers un client au départ de la commande et affichage du nom
// du client. Dans le cas d'une navigation simple, tout pourrait se formuler
// en une seule ligne.
client = (XClient)commande.clientRef.getTarget(client);
System.out.println("Client: " + client.nom);

// Navigation inverse: avec l'aide de WCollection, nous affichons maintenant toutes les
// dates de commandes passées par le client chargé.
coll = client.commandesRef.getTarget(new WCollection(), commande);
while(coll.next())
  System.out.println("Date commande: " + commande.dateCommande);

// Modification du client chargé. La propriété client.version est
// automatiquement incrémentée de 1.
client.nom.set("un nouveau nom quelconque vraiment sans importance");
client.modify();

// Suppression de ce client
client.remove();

// Création d'un produit. Les propriétés produit.id et produit.version sont garnies
// automatiquement juste avant l'exécution de l'ordre de création.
produit.code.set("WHISLOCHLO");
produit.nom.set("Whisky Loch Lomond");
produit.stock.set(10);
produit.prix.set(1599); // BEF!
produit.rayon.set(29);
produit.create();

// Conflit. La mise à jour de produit2 est refusée car produit2 ne dispose pas de la
// dernière version de l'enregistrement, puisque une mise à jour sur produit a été
// effectuée alors que produit2 avait déjà chargé le même enregistrement.
produit2.id.set(produit.id.get());
produit2.load();
```

```
produit.prix.set(1699);
produit.modify();
produit2.prix.set(1499);
produit2.modify();
```

Figure IV-31 Exemples de code Java illustrant l'utilisation des classes du noyau

IV.5.2 Gestion des commandes

Nous allons implémenter un ensemble de modules permettant une gestion élémentaire des commandes, c'est-à-dire:

- Sélection d'une commande parmi les commandes passées par un client donné.
- Visualisation du détail d'une commande
- Encodage d'une nouvelle commande
- Suppression d'une commande

En outre, les modules suivants devront être préalablement définis:

- Sélection d'un client
- Sélection d'un produit (en stock)

a) Sélection d'un client

```
// Si l'utilisateur a sélectionné un client, retourne un objet de type XClient
// représentant le client sélectionné, sinon retourne "null".
public XClient selectClient()
{
  XClient      client  = new XClient();
  WCollection  clients = new WCollection();
  WSortOrder   order   = new WSortOrder;

  clients.registerObject(client);

  order.ascending(client.nom);
  clients.setSortOrder(order);

  // Affichage des clients dans une liste de sélection à laquelle on fournit les
  // identifiants et les noms des clients.
  while(clients.next());
    _INSERT_ITEM_INTO_SELECTION_LIST_(client.id, client.nom);

  // On attend que l'utilisateur sélectionne un client ou abandonne la sélection.
  _WAIT_FOR_USER_ACTION_

  // On suppose ici que la liste de sélection est capable de fournir l'identifiant du
  // client sélectionné. Si la liste de sélection retourne "null", alors l'utilisateur
  // a annulé la sélection.
  if (_GET_SELECTED_ITEM_IDENTIFIER_ == null)
    client = null;
  else
  {
    client.id.setValue(_GET_SELECTED_ITEM_IDENTIFIER_);
    client.load();
  }
  return (client);
}
```

Figure IV-32 Module de sélection d'un client

b) Sélection d'un produit en stock

```
// Si l'utilisateur a sélectionné un produit, retourne un objet de type XProduit
// représentant le produit sélectionné, sinon retourne "null".
// Seuls les produits en stock sont sélectionnables.
public XProduit selectProduit()
{
  XProduit     produit  = new XProduit();
  WCollection  produits = new WCollection();
  WSortOrder   order    = new WSortOrder;

  produits.registerObject(produit);

  order.ascending(produit.nom);
  produits.setSortOrder(order);

  produits.setCondition(new WIsGreaterThanValue(produit.stock, 0, true);

  // Affichage des produits dans une liste de sélection à laquelle on fournit les
  // identifiants et les noms des produits.
  while(produits.next());
    _INSERT_ITEM_INTO_SELECTION_LIST_(produit.id, produit.nom);

  // On attend que l'utilisateur sélectionne un produit ou abandonne la sélection.
  _WAIT_FOR_USER_ACTION_

  // On suppose ici que la liste de sélection est capable de fournir l'identifiant du
  // produit sélectionné. Si la liste de sélection retourne "null", alors l'utilisateur
  // a annulé la sélection.
  if (_GET_SELECTED_ITEM_IDENTIFIER_ == null)
    produit = null;
  else
  {
    produit.id.setValue(_GET_SELECTED_ITEM_IDENTIFIER_);
    produit.load();
  }
  return (produit);
}
```

Figure IV-33 Module de sélection d'un produit

c) Sélection d'une commande

```
// Si l'utilisateur a sélectionné une commande, retourne un objet de type XCommande
// représentant la commande sélectionnée, sinon retourne "null".
public XCommande selectCommande(XClient client)
{
  XCommande    commande  = new XCommande();
  WCollection  commandes = new WCollection();
  WSortOrder   order     = new WSortOrder;

  commandes.registerObject(commande);
  commandes = client.commandesRef.getTarget(commandes); // Navigation inverse

  order.ascending(commande.dateLivraison);
  commandes.setSortOrder(order);

  // Affichage des commandes dans une liste à laquelle on fournit les identifiants
  // ainsi que les numéros, dates de commande et de livraison.
  while(commandes.next());
    _INSERT_ITEM_INTO_SELECTION_LIST_(commande.id, "" +
                                                  commande.numéro + " " +
                                                  commande.dateCommande + " " +
                                                  commande.dateLivraison);

  // On attend que l'utilisateur sélectionne une commande ou abandonne la sélection.
  _WAIT_FOR_USER_ACTION_

  // On suppose ici que la liste de sélection est capable de fournir l'identifiant de la
  // commande sélectionnée. Si la liste de sélection retourne "null", alors
  // l'utilisateur a annulé la sélection.
```

```
  if (_GET_SELECTED_ITEM_IDENTIFIER_ == null)
    commande = null;
  else
  {
    commande.id.setValue(_GET_SELECTED_ITEM_IDENTIFIER_);
    commande.load();
  }
  return (commande);

}
```

Figure IV-34 Module de sélection d'une commande

d) Visualisation du détail d'une commande

```
// Affiche le détail d'une commande
public void viewDétail(XCommande commande)
{
  XDétail      détail  = new XDétail();
  XProduit     produit = new XProduit();
  WCollection  détails = new WCollection();
  WSortOrder   order   = new WSortOrder;

  détails.registerObject(détail);
  détails.registerObject(produit, détail.produitRef);
  détails = commande.détailsRef.getTarget(détails); // Navigation inverse

  order.ascending(produit.nom);
  détails.setSortOrder(order);

  // Affichage dans une liste
  while(détails.next());
    _INSERT_ITEM_INTO_LIST_(détail.id, "" +
                                       produit.nom + " " +
                                       détail.quantité + " " +
                                       détail.prix);

  // On attend que l'utilisateur termine la visualisation
  _WAIT_FOR_USER_ACTION_
}
```

Figure IV-35 Module de de visualisation du détail d'une commande

e) Encodage d'une nouvelle commande

En pratique, ce module devrait se décomposer en deux sous-modules: le premier prendrait en charge la saisie à l'écran de la commande, le deuxième se chargerait de l'enregistrement dans la base de données.

```
// Si la création est réussie, retourne un objet de type XCommande représentant la
// nouvelle commande, sinon retourne "null".
public XCommande createCommande(XClient client)
{
  XCommande    commande = new XCommande();
  XDétail      détail   = new XDétail();
  WTransaction tran     = new WTransaction();

  try
  {
    tran.registerComponent(commande);
    tran.registerComponent(détail);

    // On suppose que l'utilisateur encode un numéro, ainsi que les dates de commande
    // et de livraison à l'écran
    _WAIT_FOR_USER_ACTION_

    commande.numéro.setValue(_GET_DATE_COMMANDE_);
    commande.dateCommande.setValue(_GET_DATE_COMMANDE_);
    commande.dateLivraison.setValue(_GET_DATE_LIVRAISON_);
```

```
    commande.clientId.setValue(client.id.getValue());
    commande.create();

    // Tant que l'utilisateur sélectionne un produit, on ajoute un détail à la commande

    while((produit = selectProduit()) != null)
    {
      // On suppose que l'utilisateur encode un prix et une quantité à l'écran.
      // Le prix du produit (produit.prix.getValue()) est proposé comme prix par
      // défaut.
      détail.quantité.setValue(_GET_QUANTITE_);
      détail.prix.setValue(_GET_PRIX_);
      détail.produitId.setValue(produit.id.getValue());
      détail.create();
    }

    tran.complete();

  } // try
  catch (Exception e)
  {
    tran.abort();
  }

  tran.cleanup();
}
```

Figure IV-36 Module d'encodage d'une nouvelle commande

f) Suppression d'une commande

On suppose ici que les suppressions en cascade ne sont pas gérées au niveau de la base de données.

```
// Supprime une commande et son détail
public void removeCommande(XCommande commande)
{
  XDétail      détail  = new XDétail();
  WCollection  détails = new WCollection();
  WTransaction tran    = new WTransaction();

  try
  {
    tran.registerComponent(détail);
    tran.registerComponent(détails);

    détails.registerObject(détail);
    détails = commande.détailsRef.getTarget(détails); // Navigation inverse

    détails.remove(détail);
    commande.remove();

    tran.complete();

  } // try
  catch (Exception e)
  {
    tran.abort();
  }

  tran.cleanup();
}
```

Figure IV-37 Module de suppression d'une commande

V

Langage de définition de ARROW

Le chapitre précédent nous a permis de découvrir comment la couche client du modèle pouvait répondre aux besoins fonctionnels du développeur. Pour cette raison, nous avons délibérément laissé la question suivante en suspend: "comment définir ses propres objets (spécialisations de *WObject*)?". Cette tâche est normalement dédiée à un administrateur de données et fait l'objet de ce chapitre. Le développeur, lui, exploite les objets préalablement définis et peut donc se contenter des fonctionnalités présentées dans le chapitre précédent.

V.1 Définition du noyau

Jusqu'à présent, nous avons toujours présenté les classes de la base de données illustrative "clé sur porte". En effet, les exemples que nous avons développés dans le chapitre précédent comprenait souvent des lignes de code similaires à celles énoncées dans la Figure V-1.

```
// Déclaration et création des instances de classes héritées de
// WObject pour chacune des tables de la base de données illustrative
XClient   client   = new XClient();
XCommande commande = new XCommande();
XDétail   détail   = new XDétail();
XProduit  produit  = new XProduit();
```

Figure V-1 Déclaration et création d'instances de classes qui spécialisent *WObject*

Comment les classes *XClient*, *XCommande*, *XDétail* et *XProduit* ont été définies reste un mystère que ce chapitre va dévoiler. En outre, nous allons examiner le comportement interne des composants du noyau. Il s'agit là d'une approche indispensable pour quiconque désire spécialiser les classes du noyau.

V.1.1 WProperty

a) Ressource sous-jacente

La connexion physique entre une instance de *WProperty* et le champ d'un enregistrement (la ressource) est réalisable par l'intermédiaire du nom que porte ce champ au sein de l'enregistrement. Le nom de ce champ est défini une fois pour toute et ne peut donc changer au cours de la vie d'une instance de *WProperty*. Il est disponible en consultation grâce à la méthode *getFieldName()* que nous avons présentée dans le chapitre précédent (cfr IV.1.4.c).

Modélisation
 # constructor(fieldName : String) : WProperty > Exception
Crée une instance de *WProperty* et l'associe à un nom de champ. Déclenche une exception si le nom de champ est vide. Méthode protégée car doit impérativement être appelée d'une classe spécialisée.

b) Gestion des valeurs

La définition du domaine de valeur est matérialisée sous la forme de méthodes abstraites que des classes spécialisées devront donc implémenter. La première méthode est chargée du contrôle du type de valeur et la deuxième du domaine de valeurs proprement dit. Dans le cas de l'assignation d'un entier - à une instance de *WProperty* spécialisée prenant en charge la gestion des entiers - devant se trouver dans un intervalle déterminé, la première méthode va vérifier qu'il s'agit bien d'un entier et la deuxième que l'entier se situe bien dans la fourchette définie.

Le choix d'utiliser deux méthodes pour le contrôle des valeurs est délibéré. La gestion des conditions en est la cause car il est parfois nécessaire de vérifier le type de la valeur sans en vérifier le domaine. Sans quoi certaines conditions seraient impossibles à exprimer. Ainsi, dans l'exemple de notre entier, si la fourchette de valeur admise est]a, b[, toute condition sur cet intervalle doit autoriser n'importe quelle valeur du même type que a, donc sans utiliser notre deuxième méthode qui vérifie le domaine de valeurs proprement dit. Sinon, il est impossible d'exprimer la condition >= a puisque a est lui-même exclu de l'intervalle.

Soulignons le fait que la gestion des domaines de valeurs est la principale raison qui peut dicter une éventuelle nécessité de spécialiser *WProperty*.

Modélisation
 # checkClass(value : Any) > Exception
Contrôle le type d'une valeur. Méthode abstraite. Si le type est invalide, une exception doit être déclenchée.
 # checkValue(value : Any) > Exception
Contrôle le domaine de valeurs d'une valeur. Méthode abstraite. Si le domaine de valeurs est violé, une exception doit être déclenchée.

c) Cardinalité

Seules les cardinalités [0-1] et [1-1] sont supportées. Un attribut spécifique permet donc de déterminer si le champ est requis (cardinalité[1-1]) ou non-requis (cardinalité [0-1]). La cardinalité est définie une fois pour toute et est disponible en consultation au moyen d'une méthode dédiée.
Il n'est pas obligatoire que la cardinalité d'une instance de *WProperty* soit identique à celle du champ sous-jacent. Si un champ est non-requis au niveau de la base de données, rien n'empêche l'instance de *WProperty* qui l'encapsule de spécifier ce champ comme requis. Par contre, le scénario inverse, bien que possible, présente peu d'intérêt: si le champ sous-jacent est requis, alors que l'instance de *WProperty* qui l'encapsule ne l'est pas, toute opération de mise à jour de ce champ dans la base de données avec une valeur nulle sera vouée à l'échec.

Le Tableau V-1 ci-dessous illustre les résultats envisageables lors d'une opération de mise à jour de la base de données en fonction des différentes combinaisons de cardinalité possibles:

Cardinalité du champ dans l'Instance de *WProperty*	Cardinalité du champ dans la base de données	Résultats d'une opération de mise à jour du champ avec une valeur nulle (vide)
[0-1] = Non-requis	[0-1] = Non-requis	La mise à jour est effectuée, compte tenu du fait qu'aucune autre contrainte d'intégrité n'est violée.
[0-1] = Non-requis	[1-1] = Requis	La base de données refuse la mise à jour: situation inefficiente car la cardinalité [1-1] n'est pas appliquée au niveau de l'instance de *WProperty*. Il faut donc attendre la réponse du pilote de base de données (via la couche moteur), ce qui est évidemment coûteux dans un environnement distribué.
[1-1] = Requis	[0-1] = Non-requis	La mise à jour est refusée au niveau de la couche supérieure (donc sans faire appel au pilote de base de données via la couche moteur) car l'instance de *WProperty* spécifie le champ comme requis.
[1-1] = Requis	[1-1] = Non-requis	La mise à jour est refusée au niveau de la couche supérieure pour la même raison que ci-dessus.

Tableau V-1 Combinaisons des cardinalités entre une instance de *WProperty* et du champ qu'elle encapsule

En résumé, la cardinalité la plus restrictive sera de préférence appliquée aux instances de *WProperty* afin d'optimiser l'éventuel trafic réseau et par là d'améliorer la réactivité du client.

Avant toute création ou modification d'enregistrements dans la base de données, une instance de *WObject* va préalablement déclencher un contrôle d'intégrité. Elle va entre autres vérifier la cardinalité de toutes les instances de *WProperty* qui lui sont associées. Si une d'entre elles contient une valeur nulle alors que la cardinalité est [1-1], une exception est déclenchée.

Nous avons délibérément choisi de gérer la cardinalité au moyen d'un attribut plutôt que par une spécialisation de *WProperty*. On évite ainsi de dédoubler toutes les classes héritées de *WProperty* comme l'illustrent les schémas de la Figure V-2 et la Figure V-3.

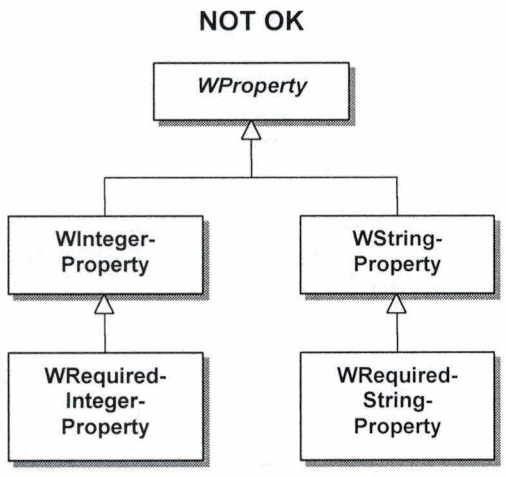

Figure V-2 Le nombre de classes est dédoublé car la cardinalité est implémentée par spécialisation. Cette option n'a pas été retenue pour notre modèle.

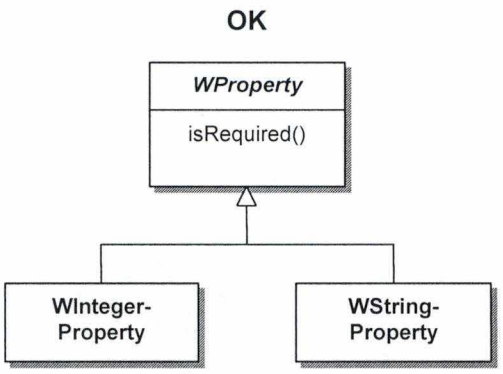

Figure V-3 La cardinalité est gérée par attribut, ce qui permet de limiter le nombre de classes et surtout de réserver la spécialisation à la gestion des domaines de valeurs. Nous avons retenu cette option pour notre modèle.

Ce choix est d'autant plus justifié que le même raisonnement pourrait être tenu à propos de la participation à l'identifiant et la gestion des cachets, que nous abordons ci-après. Dans une structure telle que proposée par la Figure V-2, nous aurions dû multiplier le nombre de classes spécialisées de *WProperty*

par huit. Ainsi, la gestion des entiers aurait requis les classes suivantes: *WIntegerProperty*, *WRequiredIntegerProperty*, *WKeyIntegerProperty*, *WStampIntegerProperty*, *WRequiredKeyIntegerProperty*, *WRequiredStampIntegerProperty*, *WKeyStampIntegerProperty* et *WRequiredKeyStampProperty*.

Modélisation
 # constructor(fieldName : String, required : Boolean) : WProperty > Exception
Crée une instance, l'associe à un nom de champ et définit sa cardinalité (requis ou non-requis). Déclenche une exception si le nom de champ est vide. Méthode protégée car doit impérativement être appelée d'une classe spécialisée.

d) Identifiant

Une instance de *WObject* doit connaître les champs de la table sous-jacente participant à l'identifiant, avant de pouvoir manipuler les enregistrements de ladite table. Plusieurs instances de *WProperty* associées à la même instance de *WObject* peuvent participer à l'identifiant, permettant ainsi de gérer la connexion aux enregistrements dont la clé est composée de plusieurs champs.

Si la table sous-jacente dispose de plusieurs clés, alors une d'entre elles doit être choisie (de préférence la clé primaire). De plus, tous les champs composant la clé choisie doivent être encapsulés chacun par une instance de *WProperty* définie comme participant à l'identifiant.

En cas de chargement, de création ou même de modification, les valeurs des champs participant à l'identifiant sont mémorisées pour être ultérieurement réutilisées comme critère de sélection lors des opérations de suppression ou de modification. En conséquence, parallèlement au stockage temporaire des valeurs, on conserve une copie supplémentaire des valeurs des champs participant à l'identifiant. La Figure V-4 illustre en pseudo-code cette nécessité de duplication.

```
// On charge un objet client
Client.Charger avec enregistrement où Id = '345'

// On change la valeur de son identifiant
Client.Id = '678'

// On met l'enregistrement sous-jacent à jour en assignant la valeur
// '678' au champ encapsulé par Client.Id, et ce pour l'enregistrement
// où le champ encapsulé par Client.Id vaut '345'
Client.Modifier
```

Figure V-4 Illustration de la nécessité de conserver une copie des valeurs de champs participant à l'identifiant. L'opération 'Modifier' nécessite l'intervention de deux valeurs pour le même champ: la nouvelle valeur (celle à mettre à jour) et l'ancienne (celle utilisée pour sélectionner l'enregistrement).

Le schéma de la Figure V-5 ci-dessous nous présente les différentes étapes intervenant dans l'interaction entre la valeur d'un champ d'un enregistrement participant à la composition de l'identifiant, et ses deux copies dans une instance de *WProperty*:

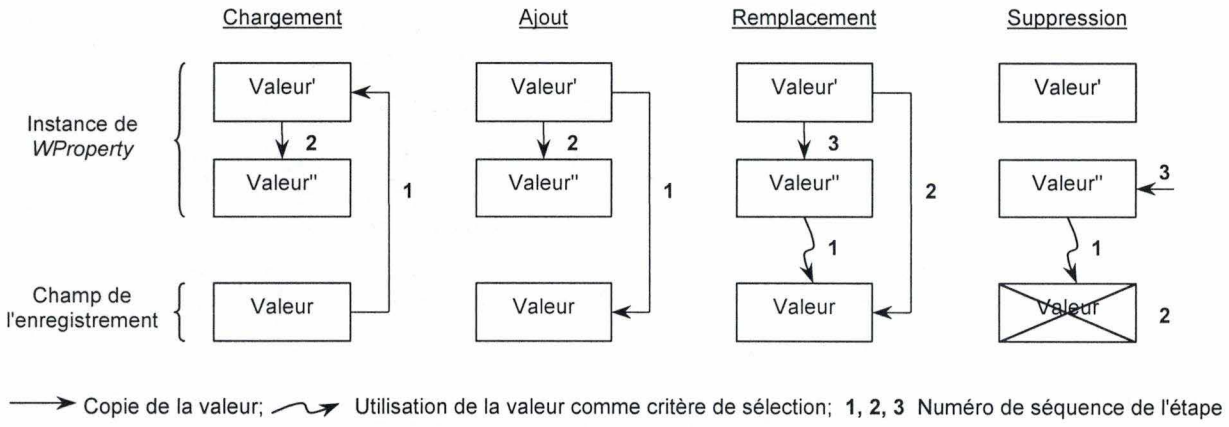

Figure V-5 Coordination des copies de valeurs d'une instance de *WProperty* avec la valeur de champ de l'enregistrement sous-jacent pour les opérations de chargement, création, modification et suppression.

On remarque que:

- *Valeur'* est utilisé comme zone mémoire temporaire contenant les valeurs à destination ou en provenance d'un enregistrement (c'est là que sont stockées les valeurs de champs). La double copie *Valeur''* est uniquement utilisée pour resélectionner l'enregistrement auquel on est connecté.
- La modification et la suppression ne sont possibles qu'à partir du moment ou *Valeur''* est garni, donc suite à un chargement ou une création. Les enchaînements possibles ont été étudiés plus haut (cfr III.2.1.b).
- Lors d'une suppression, *Valeur''* est réinitialisé à "zéro" car il n'y plus lieu de se connecter à un enregistrement supprimé.

Nous profitons du mécanisme de double copie pour étendre les fonctionnalités offertes par le couple *WProperty - WObject*. Ainsi, deux opérations complémentaires peuvent être envisagées: le rechargement et la restitution. Le rechargement permet de re-charger l'enregistrement désigné par les valeurs d'identifiant stockées dans *Valeur''*. La restitution requiert l'extension du mécanisme de double copie à toutes les intances de *WProperty* (et plus seulement aux instances encapsulant des champs participant à la clé). Il permet de récupérer les valeurs de champs telles qu'elles étaient lors du dernier (re)chargement, création ou modification. La Figure V-6 ci-dessous illustre l'interaction entre *Valeur*, *Valeur'* et *Valeur''* appliquée à nos deux nouvelles opérations.

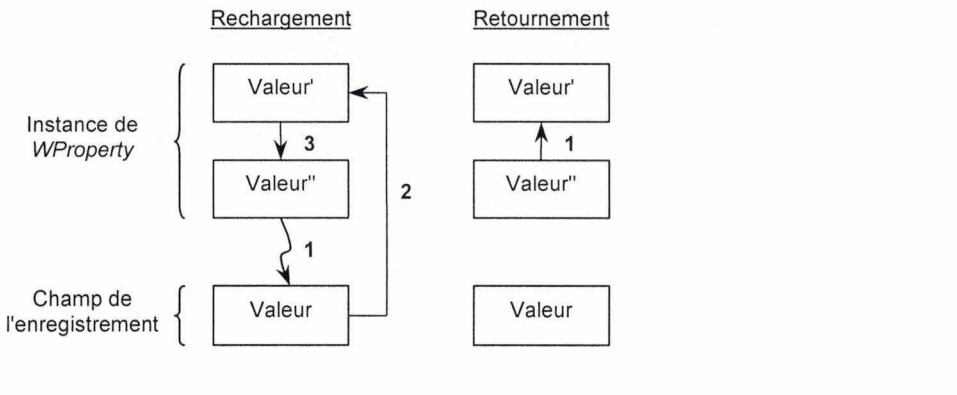

Figure V-6 Coordination des copies de valeurs d'une instance de *WProperty* avec la valeur de champ de l'enregistrement sous-jacent pour les opérations de rechargement et restitution.

Signalons que toutes ces manipulations de valeurs sont initiées et chapeautées par *WObject* et ce, pour les instances de *WProperty* qui y sont associées. Les méthodes de la classe *WProperty* dédiées à ces manipulations sont donc protégées et accessibles uniquement à partir de *WObject*.

Modélisation

+ constructor(fieldName : String, required : Boolean, key : Boolean) : WProperty > Exception
Crée une instance, l'associe à un nom de champ, définit sa cardinalité (requis ou non-requis) et la définit comme participant à la composition de l'identifiant. Déclenche une exception si le nom de champ est vide. Méthode protégée car doit impérativement être appelée d'une classe spécialisée.

 # assignSnapshot()
Copie *Valeur'* (la valeur disponible via *getValue()*) dans *Valeur''* suite à un (re)chargement, une création, ou une modification. Cette méthode est protégée car accessible seulement à partir de *WObject*.

 # revertSnapshot()
Copie *Valeur''* dans *Valeur'* lors d'une opération de restitution initiée par *WObject*. Cette méthode est protégée car accessible seulement à partir de *WObject*.

 # getSnapshot() : Any
Retourne *Valeur''*. Utilisé par *WObject* pour les opérations de rechargement, modification et suppression. Cette méthode est protégée car accessible seulement à partir de *WObject*.

 # clearSnapshot()
Vide *Valeur''*. Cette méthode est protégée car accessible seulement à partir de *WObject*.

e) Cachets(stamps)

Grâce au mécanisme de double copie vu ci-dessus, il devient aisé de proposer une gestion des cachets et, par là, des accès concurrents aux enregistrements. L'idée est d'utiliser les *Valeur''* définis comme cachets en plus de celles définies comme participant à l'identifiant pour sélectionner l'enregistrement sous-jacent lors de modifications et suppressions.

Décrivons le problème. Supposons que nous chargons un enregistrement et qu'ensuite un autre utilisateur le modifie. Dans ce cas, sans le savoir, nous ne disposons pas de la dernière version de l'enregistrement. De plus, si à notre tour nous remplaçons cet enregistrement, les modifications apportées par l'autre utilisateur seront perdues et ce, sans que quiconque n'en soit informé.
En outre, nous avons fixé dans les limites (cfr III.4) que:

- Aucun système de notification ne devait être implémenté, en conséquence de quoi il est impossible de détecter pro-activement qu'un autre utilisateur manipule ou a manipulé l'enregistrement sur lequel on est connecté.
- Les blocages d'enregistrement ne sont pas gérés.

Nous sommes donc contraints de proposer une solution intervenant à posteriori basée sur le principe des cachets, au niveau des opérations de modification et suppression.

Le champ *version* de notre base de données illustrative est un exemple de cachet. Lors du chargement de l'enregistrement, la valeur du champ *version* est recopiée dans *Valeur'* et *Valeur''* (comme c'est d'ailleurs le cas pour tous les champs). Ensuite, lors de la modification, le numéro de version de *Valeur'* est initialisé avec *Valeur''* + 1 (car à chaque modification, le numéro de version est incrémenté de 1), et le numéro de version stocké dans *Valeur''* est utilisé comme critère de sélection (conjointement aux *Valeur''* identifiantes). Ainsi, si un autre utilisateur a entre-temps effectué une modification, le champ *version* aura déjà été incrémenté de 1 et ne sera donc plus identique à *Valeur''*. La figure ci-dessus illustre ce processus au moyen d'un exemple où une opération de modification échoue, en se focalisant sur une instance de *WProperty* de type cachet.

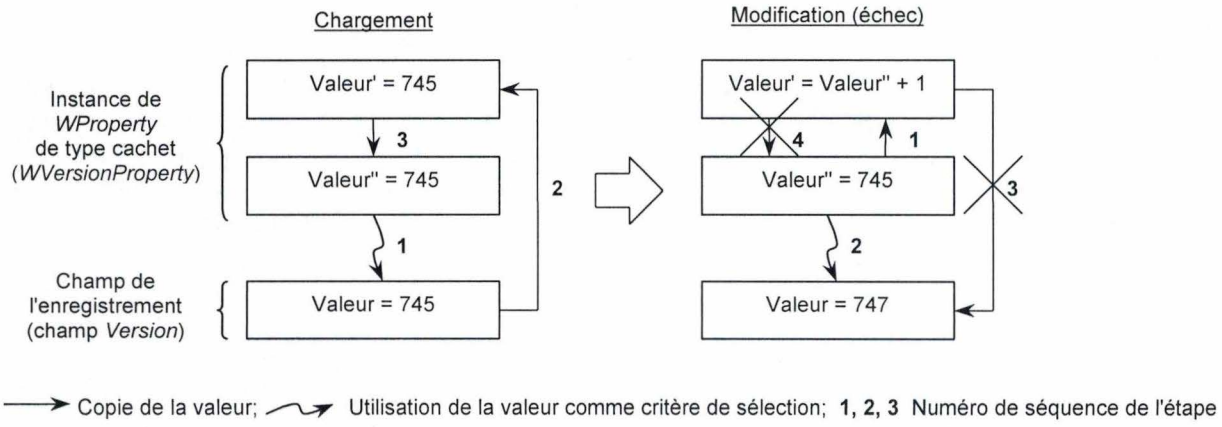

Figure V-7 Illustration de la fonction de cachet appliquée au champ *version* de la base de données illustrative. Lors de l'opération de modification, la valeur de l'enregistrement vaut *Valeur''* + 2, ce qui signifie que l'enregistrement a été modifié à deux reprises par d'autres utilisateurs depuis notre chargement. Donc la modification échoue à l'étape 2 puisque *Valeur''* est différent de *Valeur*.

Dans le cas du champ *version*, le cachet est généré par le client, puisque l'incrémentation a lieu dans une instance de *WProperty*. Si les cachets sont générés par la base de données (par exemple des timestamps), une relecture de l'enregistrement est nécessaire après toute mise à jour. Pour les suppressions, les cachets ne se préoccupent ni de la génération d'un nouveau cachet, ni de l'éventuelle relecture de l'enregistrement.

La gestion concurrentielle au moyen de cachets s'apparente en fait assez bien avec la gestion des conflits tels qu'on les rencontre dans les algorithmes de réplication.

Modélisation

+ constructor(fieldName : String, required : Boolean, key : Boolean, stamp : Boolean) : WProperty > Exception

Crée une instance, l'associe à un nom de champ, définit sa cardinalité (requis ou non-requis), la définit comme participant à la composition de l'identifiant et la définit comme cachet. Déclenche une exception si le nom de champ est vide. Méthode protégée car doit impérativement être appelée d'une classe spécialisée.

f) Événements

Une gestion d'événements élémentaire permet aux instances de *WProperty* d'être informées lorsque l'instance de *WObject* associée est sur le point d'effectuer une opération de création, de modification ou de suppression sur la base de données.
Si l'on se replace dans le contexte de l'exemple du champ *version* présenté ci-dessus, les événements vont permettre d'intercepter les intentions de création ou de modification pour assigner à *Valeur'* la valeur 1 en cas de création et la valeur *Valeur''* + 1 en cas de modification.

Modélisation

 # willCreate() > Exception

Appelé par *WObject* juste avant une opération de création. Même si cette méthode n'est pas abstraitre, son implémentation dans *WProperty* est "vide". Une exception peut être déclenchée pour avorter l'opération de création en cours. Cette méthode est protégée car appelée exclusivement par l'instance de *WObject* associée.

 # willModify() > Exception

Identique à la méthode *willCreate()* mais pour une opération de modification.

 # willRemove() > Exception

Identique à la méthode *willCreate()* mais pour une opération de suppression.

g) Transactions

En prévision de la gestion des transactions, nous rajoutons déjà certaines fonctionnalités à *WProperty*. Nous avons déjà vu que l'annulation d'une transaction provoque une réinitialisation des composants associés à cette transaction (cfr IV.3), avec la valeur qu'ils avaient au démarrage de ladite transaction. Au niveau de *WProperty*, en reprenant la notation de la Figure V-5, l'impact de l'annulation d'une transaction se situe au niveau de *Valeur'*, *Valeur''* et de l'état "altéré" ou "non-altéré". En conséquence, nous conservons une copie de ces deux valeurs et de l'état à chaque démarrage de transaction. Notons que l'association d'un composant à une transaction a un effet identique - pour ce composant - au démarrage proprement dit d'une transaction.

Modélisation

 # backup()

Sauvegarde Valeur' (la valeur disponible via *getValue()*), Valeur'' (la valeur disponible via *getSnapshot()*) et l'état (disponible via *isDirty()*). Cette méthode est protégée car utilisée exclusivement par *WObject* en réponse au démarrage d'une transaction.

 # restore()

Restitue *Valeur'*, *Valeur''* et l'état préalablement sauvegardés. Cette méthode est protégée car utilisée exclusivement par *WObject* en réponse à l'annulation d'une transaction.

h) Informations fonctionnelles

Dans les interfaces utilisateur multi-langues, il est souvent conseillé d'afficher dynamiquement les étiquettes associées aux noms de champs. De plus, une description succincte du rôle de chacun des champs est souvent appréciée (par exemple dans un tooltip). *WProperty* est l'endroit idéal pour gérer ce type d'information. Ces informations sont disponibles en consultation via les méthodes *getLabel()* et *getDescription()* présentées dans le langage de manipulation de ARROW (cfr IV.1.4. c).

Modélisation

 # setLabel(label : String)

Assigne une étiquette. Cette méthode est protégée.

 # setDescription(label : String)

Assigne une description. Idéal pour les "tooltips". Cette méthode est protégée.

i) Association à WObject

Dans le diagramme de classes du noyau (cfr Figure IV-1), toute instance de *WProperty* est associée à au moins une instance de *WObjet*. Même si une instance de *WProperty* isolée ne présente a priori aucun intérêt, il est préférable d'en autoriser l'existence. Il est ainsi possible de créer une instance de *WProperty* et ensuite de l'associer à une instance de *WObjet*.

Toutefois, une même instance de *WProperty* ne pourra être associée qu'à une et une seule instance de *WObjet*. Ceci paraît logique compte tenu du fait qu'une instance de *WProperty* ne peut encapsuler qu'un et un seul champ.

En réalité, l'implémentation de l'association [*WObjet* 1..1 - 1..* *WProperty*] exige certaines concessions. Dans la situation où une nouvelle instance de *WObjet* doit être créée et associée à une nouvelle instance de *WProperty*, soit *WObjet*, soit *WProperty* doit être instancié avant l'autre; on peut ensuite réaliser l'association. Avant que l'association ne soit réalisée, la cardinalité devient donc [*WObjet* 0..1 - 0..* *WProperty*]. Une instance de *WObject* non-associée avec au moins une instance de *WProperty* n'étant d'aucune utilité, une exception sera générée lors des tentatives de manipulation de ladite instance de *WObject*.

Modélisation

 # setParent(object : WObject) > Exception

Associe l'instance de *WProperty* courante à l'instance de *WObject* fournie en paramètre. Cette méthode est protégée car seul *WObject* peut y accéder via *WObject.addProperty()*. Une exception est déclenchée si le paramètre *object* est vide ou si l'instance de *WProperty* courante est déjà associée à une autre instance de *WObject*.

j) Modélisation

La Figure V-8 ci-dessous résume la modélisation de *WProperty*, en intégrant les méthodes propres aux langages de manipulation (méthodes publiques) et de définition (méthodes protégées).

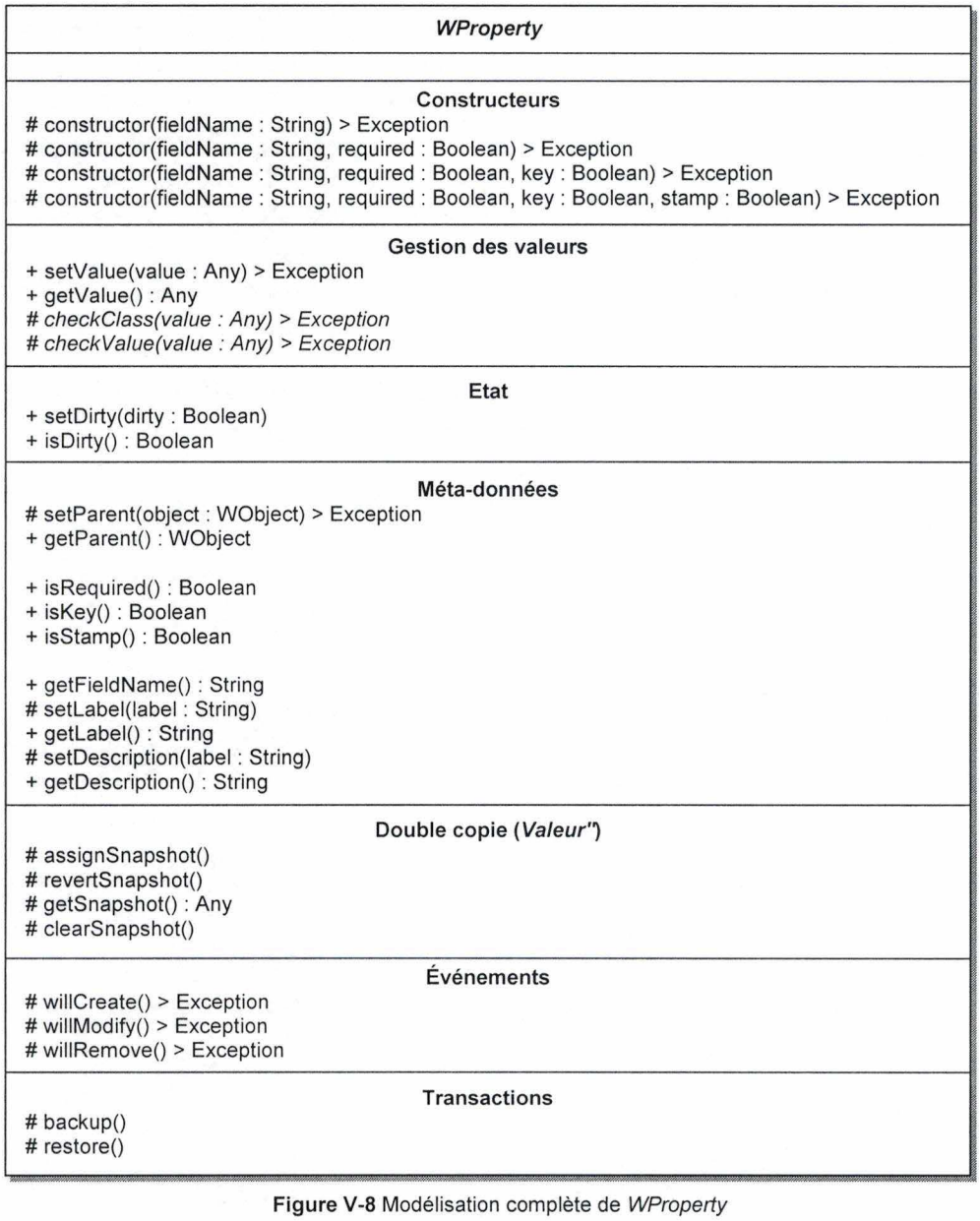

Figure V-8 Modélisation complète de *WProperty*

k) Spécialisations

Les extensions de *WProperty*, que nous avons présentées dans les extensions du noyau (cfr IV.1.2), redéfinissent certaines des méthodes protégées.
Examinons ces différentes spécialisations, en détaillant certaines d'entre elles:

- **WIntegerProperty.** Cette classe prend en charge la gestion des valeurs de type entier. Puisque toutes les valeurs entières sont acceptées, le domaine de valeurs au sein des entiers est infini, et donc seul un contrôle au niveau du type est nécessaire.

Modélisation
+ constructor(...) > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WProperty*).

 # checkClass(value : Any) > Exception
Contrôle que la valeur est bien de type entier. Déclenche une exception si ce n'est pas le cas.
 # checkValue(value : Any) > ~~Exception~~
Ne fait rien car tous les entiers sont acceptés (le domaine de valeurs n'est pas limité).
Aucune exception n'est donc déclenchable.

- **WIntervalIntegerProperty.** Cette classe prend en charge la gestion des valeurs de type entier appartenant à un intervalle déterminé. Pour ce faire, nous spécialisons *WIntergerProperty* en spécifiant un intervalle fermé qui définit le domaine de valeurs.

Modélisation
 # constructor(..., min : Integer, max : Integer) > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WProperty*) et spécifie un intervalle fermé dont les bornes sont *min* et *max*.
 # checkValue(value : Any) > Exception
Vérifie que la valeur est bien comprise dans l'intervalle défini et déclenche une exception si ce n'est pas le cas.

- **WVersionProperty.** Cette classe prend en charge la gestion des champs de type version, tels qu'on les trouve dans notre base de données illustrative. Sa valeur est initialisée à 1 lors d'une création, et est incrémentée lors de chaque modification.

Modélisation
+ constructor(...) > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WIntegerProperty*).
 # willCreate() > Exception
Copie 1 dans la valeur de l'instance (via la méthode *set()* de *WIntegerProperty*)
 # willModify() > Exception
Copie *Valeur''* (via la méthode *getSnapshot()*) + 1 dans la valeur de l'instance (dans *Valeur'*).

- **WDoubleProperty.** Cette classe prend en charge la gestion des valeurs de type réel. Sa modélisation est équivalente à *WIntegerProperty*.

Modélisation
+ constructor(...) > Exception
 # checkClass(value : Any) > Exception
 # checkValue(value : Any) > ~~Exception~~

- **WDateProperty.** Cette classe prend en charge la gestion des valeurs de type date. Sa modélisation est équivalente à *WIntegerProperty*.

Modélisation
+ constructor(...) > Exception
 # checkClass(value : Any) > Exception
 # checkValue(value : Any) > ~~Exception~~

- **WStringProperty.** Cette classe prend en charge la gestion des valeurs de type texte. Sa modélisation est équivalente à *WIntegerProperty*.

Modélisation
+ constructor(...) > Exception
 # checkClass(value : Any) > Exception
 # checkValue(value : Any) > ~~Exception~~

- **WLimitedStringProperty.** Cette classe prend en charge la gestion des valeurs de type texte dont la longueur minimale et maximale est déterminée. Sa modélisation est équivalente à *WIntervalIntegerProperty*, mais à partir d'une spécialisation de *WStringProperty*.

Modélisation
+ constructor(..., min : Integer, max : Integer) > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WProperty*) et spécifie une longueur minimale et maximale autorisée.
 # checkValue(value : Any) > Exception

- **WIdProperty.** Cette classe prend en charge la gestion des champs de type identifiant, tels qu'on les trouve dans notre base de données illustrative. Il s'agit d'un champ dont la valeur est générée aléatoirement lors des créations. La création aléatoire d'identifiants techniques est une solution performante car non seulement la base de données est déchargée de cette tâche mais en plus l'identifiant est immédiatement disponible pour le client, sans qu'il doive recharger l'enregistrement qui vient d'être créer. De plus, si chaque client est identifiable a priori (par exemple: applications intra/extra-net), on peut imaginer d'améliorer la génération des identifiants en les préfixant par une clé propre au client.

Modélisation
+ constructor(..., length : Integer) > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WLimitedStringPropertyProperty*) en spécifiant une longueur minimale et maximale de *length* caractères. *length* doit être plus grand ou égal à 10 sinon une exception est déclenchée.
 # willCreate() > Exception
Copie une chaîne de caractères générée aléaltoirement dans la valeur de l'instance (via la méthode *set()* de *WLimitedStringProperty*). Cette chaîne de caractères est une conversion en base 36 (10 chiffres + 26 lettres) de la concaténation du nombre de millisecondes écoulées depuis une date butoir et d'un nombre généré aléatoirement, pour un total de *length* caractères. Si nous réservons 10 caractères pour le stockage des millisecondes, nous disposons donc de $(36^{(length-10)} - 1)$ combinaisons pour notre génération aléatoire. La probabilité que deux clients génèrent la même clé est quasi nulle. Quand bien même cela arriverait, un des deux clients verrait alors son opération de création refusée. Un mécanisme pourrait être mis en place pour permettre plusieurs tentatives de créations consécutives en cas de conflit.

- **WStringProperty.** Cette classe prend en charge la gestion des valeurs de type quelconque, sans imposer de domaine de valeurs.

Modélisation
+ constructor(..., typeName : String) > Exception
Crée une instance (les ... remplacent les paramètres des constructeurs de *WProperty*) et spécifie le type de données à accepter.
 # checkClass(value : Any) > Exception
Contrôle que la valeur est bien du type attendu (le type fourni au constructeur).
 # checkValue(value : Any) > Exception

l) Exemples

Nous pouvons à présent définir tous les champs de notre base de données illustrative (tels que nous les avons décrits en IV.1.3), comme nous le propose la Figure V-9 ci-dessous.

```
public class XIdProperty extends WIdProperty
{
  public XIdProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_KEY | WProperty.ATTR_REQUIRED, 16); }
}

public class XVersionProperty extends WVersionProperty
{
  public XVersionProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_REQUIRED | WProperty.ATTR_STAMP); }
}

public class XForeignIdProperty extends WForeignIdProperty
{
  public XForeignIdProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0, 16); }
}
```

```
public class XCodeProperty extends WLimitedStringProperty
{
  public XCodeProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_REQUIRED, 0, 10); }
}

public class XNomProperty extends WLimitedStringProperty
{
  public XNomProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_REQUIRED, 0, 50); }
}

public class XDateProperty extends WDateProperty
{
  public XDateProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0); }
}

public class XPrixProperty extends WDoubleProperty
{
  public XPrixProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0); }
}

public class XQuantitéProperty extends WIntegerProperty
{
  public XQuantitéProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0); }
}

public class XRayonProperty extends WIntervalIntegerProperty
{
  public XRayonProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0, 1, 31); }
}
```

Figure V-9 Définition en Java des classes encapsulant les champs de la base de données illustrative.

V.1.2 WReference

a) Source et cible

Avant toute opération de navigation (via la méthode *getTarget()* que nous avons étudiée plus haut, cfr IV.1.6), il est controlé si l'instance de *WObject* cible est valide, c'est-à-dire est bien du type attendu. Ainsi, pour naviguer vers un enregistrement de la table CLIENT à partir d'une instance de *XCommande*, il est obligatoire d'utiliser une instance de *XClient*.
De plus, il faut permettre à l'instance source de déterminer vers quel enregistrement cible il faut naviguer. C'est pourquoi la classe *WReference* prévoit une méthode abstraite dont le rôle est dédié à la construction d'une condition entre l'enregistrement source et l'enregistrement cible.

Modélisation

 # checkTarget(target : WObject) > Exception
Vérifie que l'objet cible est valide. Par exemple, une classe spécialisée peut vérifier que la cible est bien une instance de *WObject* encapsulant une table particulière. Si la cible n'est pas valide, une exception est déclenchée.

 # makeCondition(target : WObject, forNavigationElseForJoin : Boolean) : WCondition > Exception
Si le paramètre *forNavigationElseForJoin* est vrai, crée une condition qui permettra à l'instance de *WReference* courante de démarrer le chargement de l'instance cible à partir de la base de données (navigation).
Si le paramètre *forNavigationElseForJoin* est faux, crée une condition qui définit une jointure entre les tables encapsulées par les instances de *WObject* source et cible. Cette fonctionnalité est utilisée par la méthode *WCollection.registerObject(object : WObject, joinReference : WReference)* pour la création de jointures. Une exception est déclenchée si la condition ne peut être construite.

b) Association à WObject

Pareillement à *WProperty*, toute instance de *WReference* doit avant tout être associée à une instance de *WObject* avant de revêtir une quelconque utilité. Les observations soulevées à propos de l'association de *WProperty* et *WObject* (cfr V.1.1.i) sont également d'application pour *WReference* et *WObject*, à la différence près qu'une instance de *WObject* peut très bien n'être associée à aucune instance de *WReference*.

Modélisation
 # setParent(object : WObject) > Exception
Associe l'instance de *WReference* courante à l'instance de *WObject* fournie en paramètre qui jouera alors le rôle d'instance source. Cette méthode est protégée car seul *WObject* peut y accéder via *WObject.registerReference()*. Une exception est déclenchée si le paramètre *object* est vide ou si l'instance de *WReference* courante est déjà associée à une autre instance de *WObject*.

c) Modélisation

La Figure V-10 ci-dessous résume la modélisation de *WReference*: Les méthodes propres au langage de manipulation sont publiques, celles propres au langage de définition (le propos de ce chapitre) sont protégées.

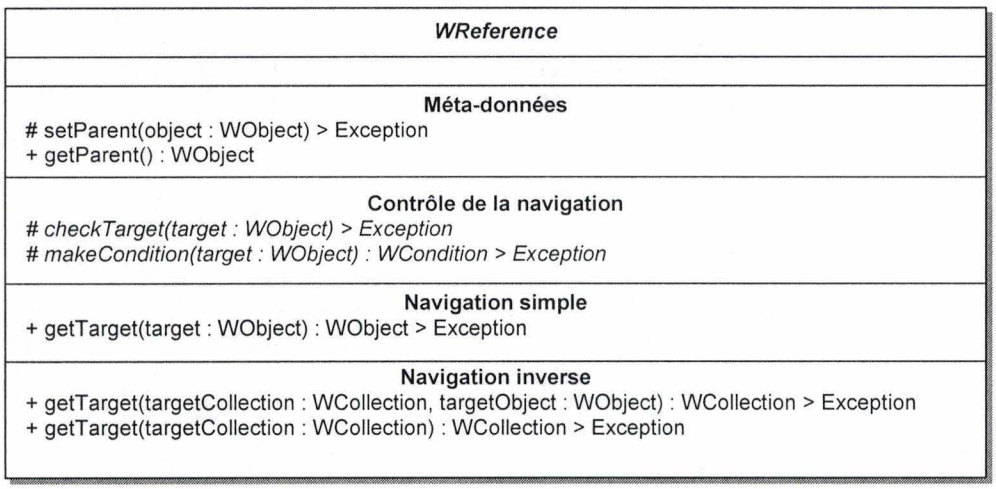

Figure V-10 Modélisation complète de *WReference*

d) Spécialisations

Notre architecture propose une seule spécialisation de *WReference*, dont le rôle est de faciliter la génération des conditions. Nous avons effectivement vu que pour permettre la navigation à partir d'une instance de *WObject*, il est nécessaire de créer un jeu de conditions sur base des identifiants et des clés étrangères des enregistrements à manipuler. De plus, si une instance de *WObject* source est implicitement accessible (car il s'agit de l'instance de *WObject* à laquelle l'instance de *WReference* est associée), il n'en va pas de même pour les instances cibles qui sont a priori inconnues. Ces soucis sont laissés aux classes spécialisant *WReference*, et en ce qui nous concerne, à *WAnyReference*.

WAnyReference offre la possibilité de définir paramétriquement:

- vers quelle instance de classe cible spécialisée de *WObject* est on censé naviguer et
- quelles sont les connexions entre les instances de *WProperty* de l'instance de *WObject* source et celles de l'instance de *WObject* cible,

Modélisation

+ **constructor(targetClassName : String) : WAnyReference > Exception**
  Crée une instance de *WAnyReference* en spécifiant le nom de la classe cible (spécialisée de *WObject*). Déclenche une exception si le nom de la classe cible est invalide (par exemple si la classe n'existe pas).
+ **constructor(targetClassName : String, sourceProperty : WProperty, targetPropertyName : String) : WAnyReference > Exception**
  Crée une instance de *WAnyReference* suivant les mêmes règles que le premier constructeur, et établit immédiatement une connexion (via la méthode *addLink()*)
+ **checkTarget(target : WObject) : WAnyReference > Exception**
  Vérifie que la classe de l'instance de *WObject* passée en paramètre correspond à celle fournie à la construction.
+ **makeCondition(target : WObject) : WCondition > Exception**
  Construit et retourne une instance de *WCondition* à partir d'une instance de *WObject* cible. Une exception est déclenchée si les attributs cibles (cfr *addLink()*) ne sont pas accessibles.
+ **addLink(sourceProperty : WProperty, targetPropertyName : String) > Exception**
  Etablit une connexion entre une instance de *WProperty* associée à l'instance de *WObject* source (donc l'instance de *WObject* auquel l'instance de *WAnyReference* courante est/sera associée) et le nom d'un attribut (en fait, d'une instance de *WProperty*) de la classe cible.
  Une exception est déclenchée si les paramètres sont invalides ou si le nom de l'attribut cible est invalide (par exemple si l'attribut n'existe pas dans la classe cible).
+ **countLinks() : Integer**
  Retourne le nombre de connexions, c'est-à-dire, dans le cas d'une navigation simple, le nombre de clés étrangères nécessaires pour pouvoir naviguer vers la cible.
+ **getProperty(n : Integer) : WProperty > Exception**
  Retourne la *n*ième instance de *WProperty* connectée, ou déclenche une exception si *n* dépasse le nombre d'instances de *WProperty* connectées.
+ **getTargetPropertyName(n : Integer) : String > Exception**
  Retourne le *n*ième nom d'attribut cible connecté, ou déclenche une exception si *n* dépasse le nombre de noms d'attributs cibles connectés.
+ **getTargetClassName() : String**
  Retourne le nom de la classe cible fournie au constructeur.

e) Exemples

Bien qu'il eut été possible de créer une classe spécialisée pour chaque type de navigation dans notre base de données illustrative, nous avons choisi d'utiliser telle quelle la classe *WAnyReference* afin de limiter le nombre de classes et par là d'éviter de complexifier inutilement les exemples.
Les exemples relatifs à *W(Any)Reference* sont présentés plus loin conjointement aux exemples de *WObject* (cfr V.1.3.g).

V.1.3 WObject

a) Ressource sous-jacente

Une instance de *WObject* établit une relation avec la ressource sous-jacente (une table ou une vue) au moyen de son nom physique. Celui-ci est donc indispensable et doit rester disponible pendant toute la durée de vie de l'instance. Notons qu'il ne s'agit pas d'établir ici une connexion à la base de données mais seulement de spécifier un nom de table.
De plus, puisque *WObject* est destiné avant tout à effectuer des opérations sur la base de données, il est naturel qu'il communique avec le moteur de ARROW. C'est pourquoi chaque instance de *WObject* doit obligatoirement disposer à tout moment d'une instance de *WArchiver* qui est le point d'encrage de la couche client dans la couche moteur: tout ordre d'opération sur la base de données initiée par la couche client transite par une instance de *WArchiver* avant d'être transmis à la base de données elle-même. *WArchiver* étant étudié plus loin (cfr VI), nous nous bornons ici à considérer qu'il s'agit d'un composant indispensable au bon fonctionnement de *WObject*.

Modélisation

 # constructor(archiver : WArchiver, tableName : String) : WObject > Exception
Crée une instance de *WObject*, l'associe à un archiveur et à un nom de table (ou de vue).
Déclenche une exception si l'archiveur ou si le nom de table est vide.

b) Aggrégation de WProperty et WReference

Une instance de *WObject* n'est d'aucune utilité si elle n'est pas associée avec au moins une instance de *WProperty*, puisque l'encapsulation des champs proprement dite est gérée par *WProperty*. L'association entre ces deux classes a déjà été discutée (cfr V.1.1.i) et V.1.2.b)).
Il est obligatoire qu'au moins une instance de *WProperty* associée soit définie comme participant à la composition de l'identifiant, afin de pouvoir sélectionner les enregistrements à manipuler.
Chaque instance de *WObject* peut également disposer d'une ou plusieurs instances de *WReference*.

En outre, une méthode de *WObject* est dédiée à l'automatisation de l'association d'instances de *WProperty* et *WReference*. Son rôle est de parcourir les attributs (les données membres) publics d'une classe héritée de *WObject*, et de systématiquement s'associer à ceux dont le type est un descendant de *WProperty* ou *WReference*.

Enfin, une méthode permet d'empêcher d'associer des instances complémentaires de *WProperty* et *WReference*. Il s'agit en réalité d'une méthode dont le rôle est de figer la définition des méta-données. Elle concerne donc également la définition des étiquettes et descriptions, ainsi que toutes les méta-données des instances de *WProperty* et *WReference* déjà associées.

Modélisation

 # registerProperty(property : WProperty) > Exception
Associe une instance de *WProperty*. Une exception est déclenchée (via la méthode *setParent()* de *WProperty*) si l'instance de *WProperty* est déjà associée ou si elle est vide

 # registerReference(reference : WReference) > Exception
Associe une instance de *WReference*. Une exception est déclenchée (via la méthode *setParent()* de *WReference*) si l'instance de *WReference* est déjà associée ou si elle est vide

 # register()
S'associe à tous les attributs publics de type *WProperty* ou *WReference* (via les méthodes *registerProperty()* et *registerReference()* de *WObject*). Cette méthode est protégée car ne peut être appelée que par les classes spécialisées.

 # fix()
Une fois appelée, cette méthode empêche tout appel ultérieur à *register()*, *registerProperty()*, *registerReference()*, *setLabel()* et *setDescription()* de l'instance de *WObject* courante, ainsi que l'appel aux méthodes *setParent()*, *setLabel()* et *setDescription()* des instances de *WProperty* associées, et à la méthode *setParent()* des instances de *WReference* associées, et à toute autre méthode dédiée à la définition des objets (comme par exemple la méthode *WAnyReference.addLink()*).

c) Intégrité

Avant toute opération de modification d'enregistrement, il est vérifié que les valeurs stockées dans les instances de *WProperty* associées respectent bien les règles d'intégrité intra-enregistrement.

Modélisation

 # checkIntegrity() > Exception
Vérifie l'intégrité intra-enregistrement (mais pas intra-champ). Déclenche une exception si celle-ci est violée. L'implémentation par défaut vérifie si les valeurs de champs des instances de *WProperty* requises sont bien remplies. Cette méthode doit être spécialisée si des règles d'intégrité plus spécifiques doivent être développées.

d) Transactions

Tout comme pour *WProperty*, nous incorporons à *WObject* les fonctionnalités requises pour la gestion des transactions. Le comportement de WObject dans un contexte transactionnel a déjà été étudié plus haut (cfr IV.3)

Modélisation

 # backup()
Sauvegarde l'état de l'instance de *WObject* courante et toutes les instances de *WProperty* associées (via la méthode *backup()* de *WProperty*). Cette méthode est protégée car utilisée exclusivement par *WObject* en réponse à l'annulation d'une transaction.

 # restore()
Restitue l'état de l'instance de *WObject* courante et toutes les instances de *WProperty* associées (via la méthode *restore()* de *WProperty*). . Cette méthode est protégée car utilisée exclusivement par *WObject* en réponse à l'annulation d'une transaction.

e) Informations fonctionnelles

Similairement à *WProperty*, chaque instance de *WObject* peut se voir attribuer une étiquette et une description.

Modélisation
 # setLabel(label : String)
Assigne une étiquette.
 # setDescription(label : String)
Assigne une description. Idéal pour les "tooltips".

f) Modélisation

La Figure V-11 ci-dessous résume la modélisation de *WObject*. Comme pour *WProperty* et *WReference*, les méthodes protégées sont issues du langage de définition.

Figure V-11 Modélisation de *WObject*

g) Exemples

La Figure V-12 propose une définition Java de notre base de données illustrative, sur base de la modélisation que nous avions présentée en introduction du chapitre consacré au langage de manipulation (cfr IV.1.3).

```
public abstract class XObject extends WObject
{
  // Création d'un archiveur (cfr section suivante, V.2 Connexion au moteur)
  static private WArchiver moArchiver = new WSunOdbcJdbcArchiver("jdbc:odbc:Arrow");

  // Encapsulation des champs communs à toutes les tables
  final public XIdProperty      id      = new XIdProperty("ID");
  final public XVersionProperty version = new XVersionProperty("VERSION");

  // Constructeur
  public XObject(String tableName)
  { super(moArchiver, tableName); }
}


final public class XClient extends XObject
{
  // Encapsulation des champs de la table CLIENT
  final public XCodeProperty code = new XCodeProperty("CODE");
  final public XNomProperty  nom  = new XNomProperty("NOM");

  // Gestion de la navigation
  final public WAnyReference commandesRef = new WAnyReference("arrow.database.XCommande",
                                                              id, "clientId");

  // Constructeur
  public XClient()
  {
    super("CLIENT");
    register();
    fix();
  }
}


final public class XCommande extends XObject
{
  // Encapsulation des champs de la table CLIENT
  final public XForeignIdProperty clientId      = new XForeignIdProperty("CLIENT_ID", true);
  final public XCodeProperty      numéro        = new XCodeProperty("NUMERO", true);
  final public XDateProperty      dateCommande  = new XDateProperty("DATE_COMMANDE", true);
  final public XDateProperty      dateLivraison = new XDateProperty("DATE_LIVRAISON", true);

  // Gestion de la navigation
  final public WAnyReference clientRef  = new WAnyReference("arrow.database.XClient",
                                                            clientId, "id");
  final public WAnyReference détailsRef = new WAnyReference("arrow.database.XDétail", id,
                                                            "commandeId");

  // Constructeur
  public XCommande()
  {
    super("COMMANDE");
    register();
    fix();
  }

  // Contrôle la validité des données.
  protected void checkIntegrity()
  {
    super.checkIntegrity();

    if (dateLivraison.get() != null &&
        dateLivraison.get().compareTo(dateCommande.get()) < 0)
      throw(new XIntegrityException(XIntegrityException.TYPE_LIVRAISON_COMMANDE, this));
  }
}
```

```
final public class XDétail extends XObject
{
  // Encapsulation des champs de la table CLIENT
  final public XForeignIdProperty commandeId  = new XForeignIdProperty("COMMANDE_ID", true);
  final public XForeignIdProperty produitId   = new XForeignIdProperty("PRODUIT_ID", true);
  final public XQuantitéProperty  quantité    = new XQuantitéProperty("QUANTITE", true);
  final public XPrixProperty      prix        = new XPrixProperty("PRIX", true);

  // Gestion de la navigation
  final public WAnyReference commandeRef = new WAnyReference("arrow.database.XCommande",
                                                             commandeId, "id");
  final public WAnyReference produitRef  = new WAnyReference("arrow.database.XProduit",
                                                              produitId, "id");

  // Constructeur
  public XDétail()
  {
    super("DETAIL");
    register();
    fix();
  }
}


final public class XProduit extends XObject
{
  // Encapsulation des champs de la table CLIENT
  final public XCodeProperty     code  = new XCodeProperty("CODE");
  final public XNomProperty      nom   = new XNomProperty("NOM");
  final public XQuantitéProperty stock = new XQuantitéProperty("STOCK", true);
  final public XPrixProperty     prix  = new XPrixProperty("PRIX", true);
  final public XRayonProperty    rayon = new XRayonProperty("RAYON", false);

  // Gestion de la navigation
  final public WAnyReference détailsRef = new WAnyReference("arrow.database.XProduit", id,
                                                            "produitId");

  // Constructeur
  public XProduit()
  {
    super("PRODUIT");
    register();
    fix();
  }
}
```

Figure V-12 Définition en Java des classes encapsulant la base de données illustrative

Remarque: On sécurise la définition des classes *XClient, XCommande, XDétail* et *XProduit* en bloquant l'accès à leurs attributs membres en lecture seule et en les définissant comme classes terminales (il est impossible de les spécialiser) via le spécificateur Java *final*.

V.2 Connexion au moteur

A plusieurs reprises, nous avons évoqué la nécessité de disposer d'une connexion entre la couche client et la couche moteur de notre architecture. La classe *WArchiver* remplit ce rôle. Elle permet aux classes telles que *WObject*, *WCollection* et *WTransaction* d'exécuter leurs ordres sur la base de données sous-jacente. A cet effet, elle publie une interface composée de méthodes abstraites que nous décrirons dans le chapitre suivant (cfr VI). Ces méthodes ne font en effet pas partie du langage de ARROW, mais bien du langage du moteur de ARROW, que seule la couche client est abilitée à utiliser. Ce langage est normalement inaccessible au développeur.

Nous allons donc ici nous contenter de définir *WArchiver* comme une classe abstraite dont le contenu (les méthodes) est tenu secret. Il est indispensable d'instancier une classe concrète héritée de *WArchiver* pour pouvoir exploiter les classes de la couche client.

ARROW propose une classe concrète spécialisée de *WArchiver* (*WSunOdbcJdbcArchiver*) comme l'illustre la Figure V-13. Sur cette figure, on observe également la relation d'association avec *WObject*, *WCollection*, *WTransaction* et *WTransactionComponent*. Pour *WCollection* et *WTransaction*, l'association à *WArchiver* est dérivée respectivement de *WObject* et *WTransactionComponent*.

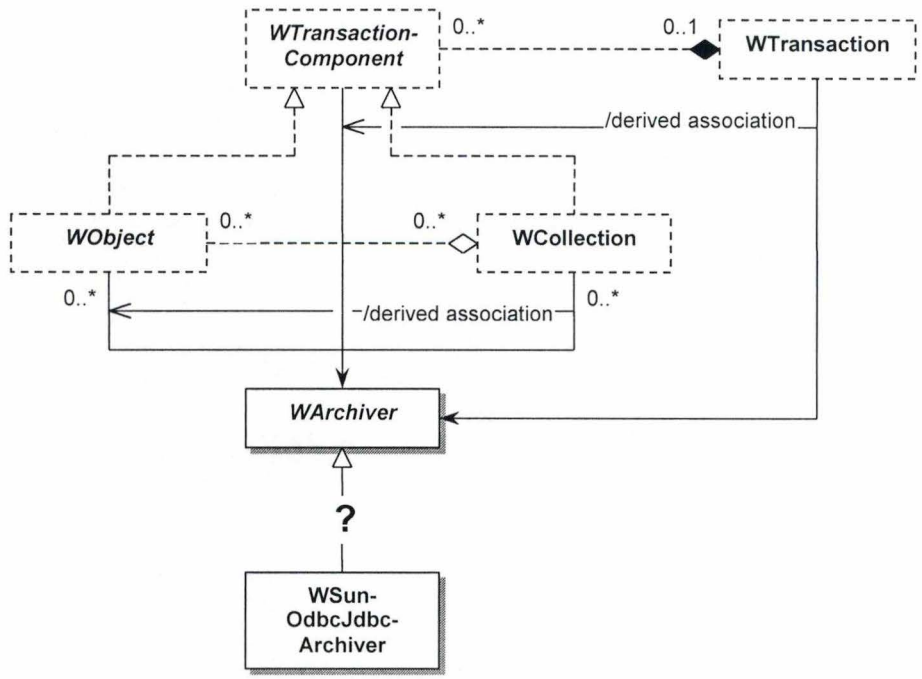

Figure V-13 Aperçu de *WArchiver* et de son intégration dans le modèle. Le "?" témoigne que l'héritage n'est pas direct, comme nous le verrons dans le chapitre suivant (cfr VI)

WSunOdbcJdbcArchiver offre au modèle la possibilité de se connecter à n'importe quelle base de données accessible par l'entremise d'un pilote ODBC. Ainsi nous avons vu précédemment (Figure V-12) dans la définition des classes encapsulant la base de données illustrative que la classe *XObject* comporte une instanciation statique de *WSunOdbcJdbcArchiver* équivalente à celle présentée ci-dessous dans la Figure V-14.

```
...

// Création d'un archiveur
static private WArchiver archiver = new WSunOdbcJdbcArchiver("jdbc:odbc:Arrow");

...
```

Figure V-14 Exemple (extrait de la Figure V-12) de connexion à une base de données via une instance de *WSunOdbcJdbcArchiver*, en spécifiant un DSN (ODBC Data Source Name) dans l'URL.

Les connexions entre l'instance de *WArchiver* et le pilote JDBC sont invisibles à la couche client. Il est toutefois intéressant de savoir que

- Un puits (pool) de connexion permet de minimiser la surcharge indue par la création et la fermeture des connexions.
- Un client consommant n transaction(s) en parallèle requerra la réservation de n connexion(s) vers le pilote JDBC.
- Une connexion vers le pilote JDBC est ouverte (ou réservée du puits) lors de la première opération sur la base de données et pas lors de la création d'une instance de *WSunObdcJdbcArchiver*.

La modélisation de *WSunObdcJdbcArchiver* se résume à:

Modélisation

+ constructor(url : String) >Exception
Crée une instance de *WSunObdcJdbcArchiver* en spécifiant la base de données à laquelle cette instance est abilitée à se connecter.

+ constructor(url : String, user : String, password : String) >Exception
Idem que pour le premier constructeur, mais en spécifiant en plus un nom d'utilisateur et un mot de passe pour permettre la connexion à la base de données.

VI
Moteur de ARROW

Dans les deux chapitres précédents, nous avons spécifié les langages de manipulation et de définition de ARROW, c'est-à-dire le langage de la couche client. Nous allons à présent examiner le langage du moteur, autrement dit, l'interface mise à la disposition de la couche client. En fait, ce chapitre décrit le "comment ça marche" et présente en sus les bases nécessaires à la conception de moteurs spécifiques.

VI.1 Aperçu général

Nous trouvons dans le moteur l'ensemble des fonctionnalités qui permettent de rencontrer les besoins de l'interface client. L'architecture du moteur s'articule autour de deux classes abstraites, *WArchiver* et *WIterator* qui représentent par ailleurs le seul point d'entrée de la couche client dans la couche moteur (cette couche moteur pouvant se situer dans un environnement d'exécution distant). En d'autres termes, ces deux classes définissent le langage du moteur. Toutes les autres classes font partie de l'implémention JDBC qui offre à la couche client la possibilité de se connecter aux bases de données relationnelles disposant d'un pilote JDBC. La Figure VI-1 illustre cette architecture.

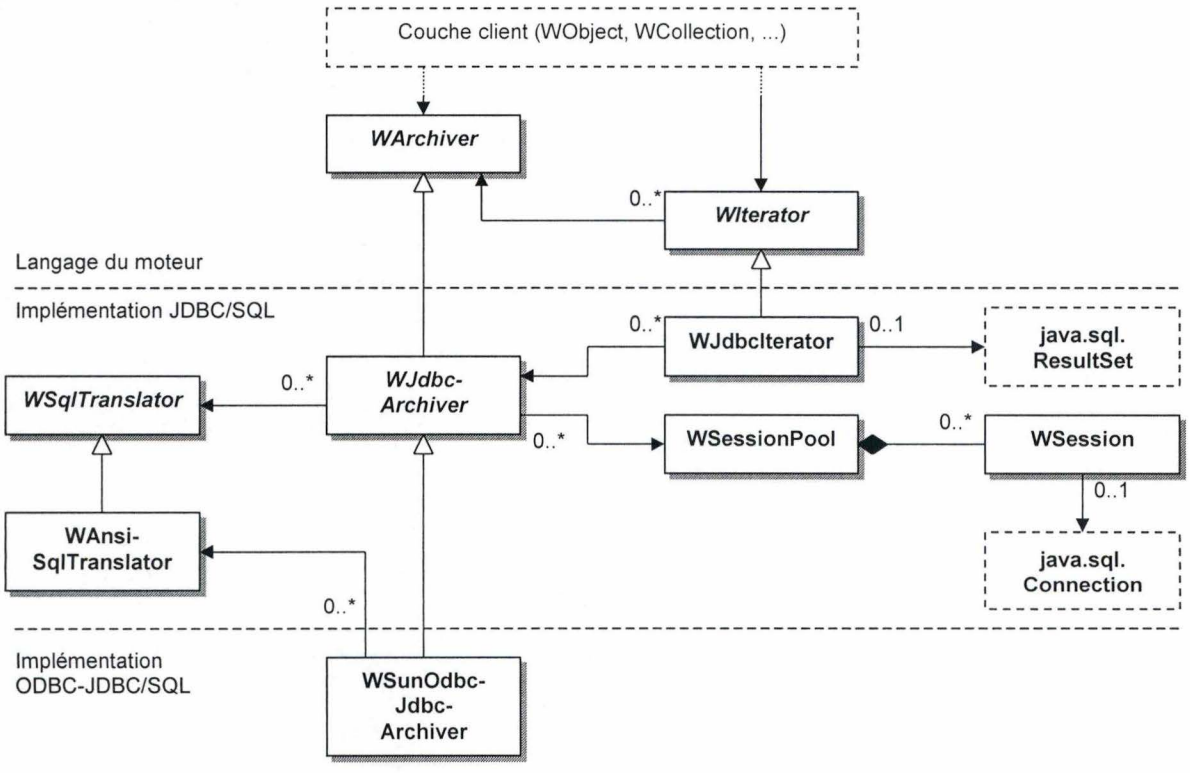

Figure VI-1 Architecture du Moteur de ARROW
et de son implémentation JDBC/SQL

Dans un premier temps, nous allons uniquement nous intéresser aux classes *WArchiver* et *WIterator*, et examiner comment elles répondent aux attentes de la couche client. Ensuite, nous examinerons succinctement l'implémentation JDBC/SQL.

VI.2 Langage du moteur

Dans le Tableau VI-1, nous énumérons les méthodes de l'interface client qui ont recours aux services de couche moteur. Les autres méthodes (non présentées dans ce tableau) sont exécutables au sein de la couche client, donc sans devoir faire appel au moteur.

Méthode de l'interface client	Action à réaliser sur la base de données	Méthode du moteur
Consultation		
WObject.load(...) *WReference.getTarget(...)*	Chargement d'un seul enregistrement	*WArchiver.load(...)*
WCollection.next(...)	Chargement de l'enregistrement suivant l'enregistrement courant d'une collection	*WIterator.next(...)*
WCollection.isLast(...)	Information: l'enregistrement courant est-il le dernier enregistrement disponible dans la collection?	*WIterator.hasNext(...)*
WCollection.count(...)	Comptage du nombre d'enregistrements dans une collection	*WArchiver.count(...)*
Mise à jour		
WObject.create(...)	Création d'un enregistrement	*WArchiver.create(...)*
WObject.modify(...) *WCollection.modify(...)*	Modification d'un ou plusieurs enregistrements	*WArchiver.modify(...)*
WObject.remove(...) *WCollection.remove(...)*	Suppression d'un ou plusieurs enregistrements	*WArchiver.remove(...)*
Transactions		
WTransaction.abort(...)	Annulation d'une transaction	*WArchiver.abort(...)*
WTransaction.complete(...)	Validation d'une transaction	*WArchiver.complete(...)*

Tableau VI-1 Méthodes de la couche client qui requièrent les services de la couche moteur

Les actions présentées dans le tableau ci-dessus sont définies dans les classes *WArchiver* et *WIterator*, sous la forme de méthodes abstraites.

WIterator prend en charge la gestion des accès à la base de données requis par les méthodes *WCollection.next()* et *isLast()*. Typiquement, une application ne sera amenée à manipuler qu'une seule instance de *WArchiver* simultanément. Par contre, chaque instance de *WCollection* est connectée à une instance

de *WIterator*, d'où la cardinalité 0..* de *WArchiver* vers *WIterator* (cfr Figure VI-1). En fait, *WIterator* implémente les fonctionnalités de base du concept de curseur pour répondre aux besoins de *WCollection*.

Plusieurs transactions parallèles étant possibles (une par instance de *WTransaction*), un identificateur permet de déterminer à quelle transaction une opération est associée. Ce même identificateur est utilisé lors de la validation ou l'annulation d'une transaction. Il est garanti unique pour toutes les instances de *WTransaction* connectées à la même instance de *WArchiver*. Si l'identificateur de transaction est "vide", l'opération n'est pas considérée comme participant à une transaction. Celle-ci est donc immédiatement validée.
Si une instance de *WObject* est associée à une instance de *WTransaction*, alors un identificateur de transaction est transmis au *WArchiver* lors des opérations de mise à jour. Dans le cas contraire, l'identificateur "vide" est utilisé.
L'utilisation d'un identificateur de transaction plutôt que la référence vers une instance de *WTransaction* permet d'éviter la surcharge ("overhead") engendrée par le tranfert d'instances entre environnements d'exécution distants.

Modélisation de WArchiver

 # load(object : WObject, condition : WCondition) : WObject > Exception
Charge un et un seul enregistrement dans une instance de *WObject*, à partir des critères fournis dans une instance de *WCondition*. L'instance de *WObject* fournie en paramètre est retournée à l'appelant.
Déclenche une exception (et ne charge donc aucun enregistrement) si aucun ou plus d'un enregistrement répond aux critères de sélection.

 # count(objects : WObject [ ], condition : WCondition) : Integer > Exception
Retourne le nombre d'enregistrements répondant aux critères fournis via une instance de *WCondition* et constitués des tables encapsulées par les instances de *WObject* fournies en paramètre.

 # create(object : WObject, transactionIdentifier : String) : Integer> Exception
Crée un enregistrement dans la table encapsulée par une instance de *WObject*, au moyen des valeurs de champs stockées dans cette instance (en fait, dans les instances de *WProperty* associées).
L'identifiant de transaction permet de déterminer à quelle transaction la création doit être incorporée. S'il est vide, alors la création est immédiatement validée.
Retourne le nombre d'enregistrements affectés par la création.

 # modify(object : WObject, condition : WCondition, transactionIdentifier : String) : Integer > Exception
Modifie un ou plusieurs enregistrements dans la table encapsulée par une instance de *WObject*, sélectionnés sur la base de conditions, au moyen des valeurs de champs stockées dans cette instance.
L'identifiant de transaction permet de déterminer à quelle transaction la modification doit être incorporée. S'il est vide, alors la modification est immédiatement validée.
Retourne le nombre d'enregistrements affectés par la modification.

 # remove(object : WObject, condition : WCondition, transactionIdentifier : String) : Integer > Exception
Supprime un ou plusieurs enregistrements dans la table encapsulée par une instance de *WObject*, sélectionnés sur la base de conditions.
L'identifiant de transaction permet de déterminer à quelle transaction la suppression doit être incorporée. S'il est vide, alors la suppression est immédiatement validée.
Retourne le nombre d'enregistrements affectés par la suppression.

 # complete(transactionIdentifier : String) > Exception
Valide toutes les opérations sur la base de données effectuées dans le cadre de la transaction dont l'identificateur est fourni en paramètre.

 # abort(transactionIdentifier : String) > Exception
Annule toutes les opérations sur la base de données effectuées dans le cadre de la transaction dont l'identificateur est fourni en paramètre.

 # createIterator(objects : WObject [ ], condition : WCondition, sortOrder : WSortOrder, maxNexts : Integer, positionnable : Boolean) > Exception
Crée une instance de *WIterator* associée à l'instance courante de *WArchiver*. Seules les valeurs des champs encapsulés par les instances de *WObject* seront rapatriés de la base de données, sous la forme d'enregistrements sélectionnés sur la base de conditions et selon un ordre donné. Le nombre maximum d'enregistrements à rapatrier peut être défini dans le paramètre *maxNexts*.
L'itérateur construit peut être repositionnable. Cette caractéristique est nécessaire lorsque l'itérateur est créé à partir d'une instance de *WCollection* participant à une transaction puisque l'itérateur doit être capable de se repositionner sur l'enregistrement qui était courant lors du démarrage de cette transaction. Le caractère positionnable est paramétrisable pour des raisons d'optimisation: lorsque l'itérateur ne participe pas à une transaction, il ne recevra aucun ordre de repositionnement. Dans ce cas, les structures à mettre en place par les classes spécialisées peuvent être moins lourdes (par exemple: un curseur *forwardonly*)

Modélisation de WIterator
 # next() : WObject [ ] > Exception
Si l'itérateur vient d'être créé, lit le premier enregistrement sinon lit l'enregistrement suivant.
Retourne les instances de *WObject* contenant les valeurs de champs de l'enregistrement venant d'être lu, ou "vide" si aucun enregistrement suivant n'est disponible.
 # hasNext() : Boolean > Exception
Retourne *vrai* s'il existe un enregistrement suivant.
 # setPosition(position : Integer) > Exception
Repositionne l'itérateur sur un enregistrement déterminé par un numéro de position. Cette méthode échoue, c'est-à-dire déclenche une exception, si l'itérateur n'a pas été créé positionnable (cfr *createIterator()*)

Les classes *WArchiver* et *WIterator* que nous venons de parcourir disposent de méthodes complémentaires dont la raison d'être est purement technique. Le lecteur curieux pourra les découvrir à travers le code source Java (cfr Annexes).

VI.3 Implémentation JDBC/SQL

L'implémentation JDBC permet d'interfacer le langage défini par les classes *WArchiver* et *WIterator* à l'API (Application Program Interface) de JDBC. Comme nous allons le découvrir ci-après, l'accent a été porté sur la performance et l'extensibilité.

a) Spécialisations de WArchiver et WIterator

Les classes *WJdbcArchiver* et *WJdbcIterator* prennent en charge la création et l'initialisation des structures JDBC nécessaires à l'exécution des requêtes, pour ensuite exécuter ces requêtes proprement dites et retourner les résultats à la couche client.

La connection à un pilote JDBC n'est pas réalisée au sein de ces classes mais est laissée au soin de classes spécialisées telles que *WSunOdbcJdbcArchiver* qui permet de créer des connexions vers le "bridge" ODBC fourni par Sun. Chaque connexion ouverte est placée dans un puits ("pool") de connexions pour permettre sa réutilisation ultérieure.

En outre, la traduction des ordres de chargement, lecture, ... en SQL est effectuée dans une classe spécifique (*WSqlTranslator*) que nous examinons plus loin (cfr Traduction SQL). C'est aux classes spécialisées de *WJdbcArchiver* de fournir le traducteur SQL à utiliser.

Modélisation de WJdbcArchiver

... implémentation des méthodes abstraites de *WArchiver* ...

- connexionPool : WSessionPool
Puits de connexion. Attribut membre privé et statique ("class-level attribute").
 # getSqlTranslator() : WSqlTranslator
Retourne le traducteur SQL à utiliser.
 # createConnection() : java.sql.Connection
Retourne une nouvelle instance de java.sql.Connection

Modélisation de WJdbcIterator

... implémentation des méthodes abstraites de *WIterator* ...

Modélisation de WSunOdbcJdbcArchiver
+ constructor(url : String) : WSunOdbcJdbcArchiver
Crée une instance de *WSunOdbcJdbcArchiver* en fournissant une url définissant le chemin d'accès vers une base de données (cfr *createConnection()*).
+ constructor(url : String, user : String, password : String) : WSunOdbcJdbcArchiver
Crée une instance de *WSunOdbcJdbcArchiver* en fournissant une url définissant le chemin d'accès vers une base de données avec un nom d'utilisateur et un mot de passe (cfr *createConnection()*).
 # getSqlTranslator() : WSqlTranslator
Retourne une instance de *WAnsiSqlTranslator*.

 # createConnection() : java.sql.Connection
Retourne une nouvelle instance de java.sql.Connection ouverte via le pilote sun.jdbc.odbc.JdbcOdbcDriver, et en utilisant l'url fournie au constructeur, ainsi que les éventuels nom d'utilisateur et mot de passe.

b) Pooling des connexions

La nécessité de disposer d'un puits (pool) de connexions vers la base de données a été dicté par les raisons suivantes:

- **Optimisation.** L'objectif est de limiter le nombre d'ouvertures et de fermetures de connexions vers la base de données, afin d'optimiser le temps nécessaire à l'exécution des requêtes et, par là, d'améliorer la réactivité de l'interface client. Le puits de connexions répond à cette attente en permettant le partage de connexions entre tous les clients connectés au même moteur, c'est-à-dire, en Java, à des instances de *WArchiver* appartenant à la même machine virtuelle. Techniquement, le puits est une variable statique de la classe *WArchiver*.
- **Gestion des transactions.** Nous avons vu qu'une transaction est démarrée par l'interface client (via une instance de *WTransaction*). Il faut donc un mécanisme permettant d'identifier la connexion associée à ladite transaction, puisque toutes les opérations effectuées sur la base de données dans le cadre de cette transaction devront transiter par la même connexion.

  Par ailleurs cette connexion devra rester ouverte et exclusivement réservée à l'instance de *WTransaction* initiatrice, tant qu'une opération d'annulation ou validation n'est pas réalisée (sous JDBC, la fermeture d'une connexion déclenche l'annulation automatique de la transaction courante).

En pratique:

- Toute connexion est placée dans le puits où elle se voit attribuer un statut booléen indiquant si elle est libre ou réservée.
- La création d'une nouvelle connexion (via la méthode *WJdbcArchiver.createConnection()*) n'est requise que si le puits ne dispose d'aucune connexion libre.
- La réservation d'une connexion est possible seulement si celle-ci est libre. La réservation d'une connexion dans le cadre d'une transaction reste effective tant que la transaction n'est pas annulée ou validée explicitement. La réservation d'une connexion hors transaction persiste le temps de l'exécution d'une opération. Par exemple, *WObject.modify()* sur une instance de *WObject* non associée à une instance de *WTransaction* bloque une connexion le temps que la modification soit exécutée.
- Toute réservation est associée à une instance de *WArchiver*. L'identifiant d'une réservation est composée de l'instance de *WArchiver* associée et de l'identificateur de transaction.
- Un mécanisme de nettoyage permet de libérer toutes les connexions réservées par une instance de *WArchiver*.
- Lorsqu'une connexion reste libre pendant un laps de temps plutôt long (paramétrisable), elle est automatiquement fermée et enlevée du puits.

Le puits de connexions est géré par les classes *WSession* et *WSessionPool*. *WSession* prend en charge la gestion unitaire des connexions, alors que *WSessionPool* gère le puits proprement dit.

Modélisation de WSession
+ release() > Exception
Libère la connexion préalablement réservée par le puits auquel appartient cette session. Déclenche une exception si la connexion n'est pas réservée.
+ isReserved() : Boolean
Retourne *vrai* si la connexion est réservée.
+ getArchiver() : WArchiver
Retourne l'instance de *WArchiver* par laquelle la connexion est réservée. Cette information n'a de sens que si la connexion est réservée.

+ getTransactionAlias() : String
Retourne l'identificateur de transaction auquel cette session est réservée. Cette information n'a un sens que si la connexion est réservée.
+ getConnection() : java.sql.Connection
Retourne la connexion.

Modélisation de WSessionPool
+ register(archiver : WArchiver, transactionIdentifier : String, connection : java.sql.Connection) : WSession > Exception
Crée une nouvelle session à partir d'une connexion, la réserve pour une instance de *WArchiver* donnée et un identificateur de transaction donné. Cette nouvelle session est ajoutée au puits. Une exception est déclenchée si la même instance de *WArchiver* associée au même identificateur de transaction a déjà une connexion réservée.
+ reserve(archiver : WArchiver, transactionIdentifier : String) : WSession
Retourne l'instance de *WSession* du puits ayant une connexion réservée par une instance de *WArchiver* donnée et un identificateur de transaction donné. Si une telle instance n'est pas trouvée, la connexion d'une instance de *WSession* quelconque (libre et appartenant au puits) est réservée pour l'instance de *WArchiver* et l'identificateur de transaction. Si aucune instance de *WSession* n'est libre dans le puits, retourne "vide".
+ find(archiver : WArchiver, transactionIdentifier : String) : WSession
Retourne l'instance de *WSession* du puits ayant une connexion réservée par une instance de *WArchiver* donnée et un identificateur de transaction donné. Retourne "vide" si une telle instance n'est pas trouvée.
+ release(archiver : WArchiver)
Libère toutes les connexions réservées par une instance de *WArchiver*.

c) Traduction SQL

Le SQL de JDBC est dépendant du type de la base de données utilisée. Pour cette raison, le moteur dispose de structures dédiées à la traduction du langage client ARROW en SQL et spécialisables en fonction du type de SQL désiré.
La classe abstraite *WSqlTranslator* définit les primitives nécessaires; la classe concrète *WAnsiSqlTranslator* propose une implémentation SQL ANSI qui, théoriquement, devrait s'adapter à tous les types de bases de données relationnelles manipulables en SQL. Ces primitives sont appelées à partir des classes spécialisées de *WJdbcArchiver*.

Modélisation de WSqlTranslator
+ getSqlLoad(object : WObject, condition : WCondition, params : java.util.Vector) : String
Crée et retourne une requête SQL de type SELECT ... WHERE ... sur une seule table.
+ getSqlLoad(objects [ ] : WObject, condition : WCondition, sortOrder : WSortOrder, params : java.util.Vector) : String
Crée et retourne une requête SQL de type SELECT ... WHERE ... ORDER BY ... sur une ou plusieurs tables.
+ getSqlCount(objects [ ] : WObject, condition : WCondition, params : java.util.Vector) : String
Crée et retourne une requête SQL de type SELECT COUNT(*) WHERE ... sur une ou plusieurs tables.
+ getSqlCreate(object : WObject, params : java.util.Vector) : String
Crée et retourne une requête SQL de type INSERT ... VALUES ... sur une seule table.
+ getSqlModify(object : WObject, condition : WCondition, params : java.util.Vector) : String
Crée et retourne une requête SQL de type UPDATE ... SET ... WHERE ... sur une seule table.
+ getSqlRemove(object : WObject, condition : WCondition, params : java.util.Vector) : String
Crée et retourne une requête SQL de type DELETE FROM ... WHERE ... sur une seule table.

Note technique concernant le paramètre "*params*" de type *java.util.Vector* : les requêtes SQL sont construites en remplaçant les valeurs par des "?" et en plaçant les valeurs proprement dites dans le vecteur *params*. Par exemple, à la requête SQL générée via *WSqlTranslator.getSqlCount()* "SELECT COUNT(*) FROM PRODUIT WHERE NOM = ? OR NOM = ?" peut correspondre le vecteur ('Eau Minérale', 'Whisky'). Il s'agit en fait de requêtes paramétriques qui sont gérées par la classe *java.sql.PreparedStatement* au sein de *WArchiver*.

Modélisation de WAnsiSqlTranslator

... implémentation des méthodes abstraites de *WSqlTranslator*...

VII
Evaluation de ARROW

La base des spécifications du modèle ARROW trouve ses racines dans les critiques soulevées lors de l'évaluation du modèle JDBC (cfr II.3). Dans ce chapitre, nous allons vérifier si notre modèle répond bien aux attentes, en le soumettant au même type d'évaluation. Ensuite nous présenterons un tableau récapitulatif confrontant les deux modèles.

VII.1 Evaluation du modèle

L'évaluation s'articule autour des mêmes critères que pour JDBC. Nous réutilisons également la même échelle de pondération, soit:

2/2 : L'évaluation du critère donne un résultat satisfaisant
1/2 : L'évaluation du critère donne un résultat médiocre
0/2 : L'évaluation du critère donne un résultat insatisfaisant

Ces valeurs sont notées entre parenthèses conjointement aux noms des critères évalués, et sont suivies d'un signe -, =, + témoignant respectivement d'une régression, d'un état identique ou d'une amélioration par rapport à JDBC.

Il est préférable d'aborder cette section parallèlement à la section II.3 présentant l'évaluation du modèle JDBC.

VII.1.1 Structure des langages

a) Diversité des langages (1.5/2 +)

Le développeur qui se contente d'exploiter des objets (instances de *WObject*) prédéfinis en fonction de la base de données à manipuler n'est plus confronté qu'au seul langage objet.
Par contre, si le développeur est chargé de définir lui-même ces objets, il fait inévitablement appel au paradigme relationnel puisqu'il est alors tenu d'appréhender des concepts tels que table, champ, relation, ...
Toutefois, dans les deux cas, le langage SQL a été abrogé.

b) Complétude des langages (1.5/2 -)

L'étude des limites du langage de ARROW (cfr IV.4) nous prouve que la palette de fonctionnalités est certainement moins riche que celle offerte par SQL. Il s'agit d'une critique en défaveur de ARROW.

Toutefois, eu égard au type d'applications auquel notre modèle est destiné (cfr I.1), nous pouvons minimiser ce manquement.

c) Risques de bogues (1/2 +)

Les risques de bogues dus à des erreurs de syntaxe, de grammaire ou de logique dans du SQL imbriqué, ou à la mauvaise exploitation de JDBC ont disparus, notamment grâce à l'uniformité des structures mises à la disposition du développeur et à la possibilité de fortement typer ("strong typing") les données manipulées.
Par contre, de nouvelles sources de bogues propres au langage de ARROW ont vu le jour. Par exemple, l'exécution de la méthode *next()* sur une instance de *WCollection* à laquelle n'est associée aucune instance de *WObject* va déclencher une exception.

d) Complexité des langages (1.5/2, +)

Contrairement à JDBC, les structures de notre modèle sont aptes à gérer les données en provenance et à destination de la base de données.
Dans la situation où le développeur se contente d'utiliser des objets (instances de *WObject*) prédéfinis, la complexité est moindre que s'il s'attèle à définir lui-même ces objets. Toutefois, cette phase de définition étant automatisable et normalement peu fréquente, sa participation à la complexité du langage est faible.
De plus, de la régression des fonctionnalités par rapport à JDBC (cfr ci-dessus, Complétude des langages) résulte une diminution de la complexité du langage ARROW.

VII.1.2 Fonctionnalités particulières

a) Transactions (2/2 +)

Les transactions du langage permettent de synchroniser les manipulations sur les variables locales et les changements dans la base de données. Plusieurs transactions peuvent être exécutées en parallèle et ce, de manière transparente, simplement en créant plusieurs instances de *WTransaction*.
Aucune sémantique n'est prévue pour les transactions imbriquées. Ceci découle du fait que notre modèle repose principalement sur JDBC.

b) Événements (0/2 =)

ARROW ne dispose d'aucune gestion événementielle, conformément à la limite posée lors des spécifications (cfr III.4.3).

c) Résolution des conflits (1.5/2 +)

La résolution des conflits est gérée a posteriori c'est-à-dire que, comme nous l'avons détaillé dans l'évaluation de JDBC, le contrôle s'effectue lors de la modification ou de la suppresion et ce, de manière transparente (cfr V.1.1.e).
Par contre, aucune sémantique n'offre la possibilité de gérer les conflits a priori, c'est-à-dire au moyen d'un mécanisme de blocage (locking).

d) Portabilité (2/2 +)

Puisque le langage de ARROW encapsule complètement JDBC (et donc SQL), et vu que son architecture modulaire permet de se connecter à différents types de pilotes d'accès aux données (et donc pas seulement JDBC/SQL), sa portabilité est quasi totale.

e) Performance (1/2 -)

Le langage répond aux critiques formulées sur la pénalité de performance occasionnée par une mauvaise utilisation de JDBC. Par contre, son manque de fonctionnalités par rapport au langage SQL (cfr plus haut, Complétude des langages) est un élément jouant en sa défaveur car il est par là nécessaire d'implémenter par programmation des fonctionnalités disponibles en natif sous SQL.
Le fait qu'une application architecturée autour de ARROW doive gérer d'office deux couches complémentaires (la couche client et la couche moteur) ne pénalise que très peu la performance.

f) Répartition (2/2)

L'architecture telle que proposée par ARROW répond aux objectifs de répartition. Une application exploitant les éléments du langage (de la couche client) peut se connecter sur une, voire plusieurs instances de couches moteur distantes.

VII.2 Tableau comparatif JDBC/ARROW

Le Tableau VII-1 confronte les caractéristiques des deux langages que nous avons étudié.

Critère	Résultat JDBC	Résultat ARROW	Commentaires JDBC	Commentaires ARROW
Structure des langages				
Diversité des langages	1.0 2	**1.5** **2**	Deux langages cohabitent (-): - objet (Java) - relationnel (SQL)	On retrouve le langage objet uniquement (+), sauf pour la définition des objets où les concepts de table, champs, ... interviennent (-). SQL disparait (+).
Complétude des langages	**2.0** **2**	1.5 2	SQL est un langage de requête complet (+)	L'encapsulation de JDBC/SQL pénalise la complétude. Toutes les expressions SQL ne sont pas reproductibles (-).
Risques de bogues	0.0 2	**1.0** **2**	Sont indétectables à la compilation: - les erreurs dans les requêtes SQL imbriquée (-), - les erreurs dues à une mauvaise visibilité et une faible vérification des types de données (-), - les erreurs logiques en général (-).	Restent uniquement les erreurs logiques d'utilisation de ARROW (+).
Complexité des langages	1.0 2	**1.5** **2**	- La grammaire SQL n'est pas toujours uniforme (-) - Les structures JDBC sont distinctes pour l'envoi et la réception de données (-) - Les préparatifs préalables à l'exécution d'une requête sont limités (+)	La grammaire est uniforme et les structures identiques pour l'envoi et la réception des données (+). Les préparatifs sont plus lourds si l'on considère la phase (sporadique) de définition des objets (-).

Fonctionnalités particulières

Transactions	1.0 2	**2.0** **2**	Les transactions sont supportées (+), mais sans imbrications (-). Deux transactions en parallèles requièrent deux connexions explicites vers la base de données (-).	Les transactions sont gérées (+), mais sans imbrication (-). Les transactions parallèles sont implicites (+), mais requièrent autant de connexions que de transactions parallèles (-).
Événements	0.0 2	0.0 2	JDBC est incapable de capter des événements (triggers) survenant en réponse à la manipulation des données (-).	Aucune sémantique n'implémente ce type d'événement (-).
Résolution des conflits	1.0 2	**1.5** **2**	La résolution des conflits doit être explicitement programmée (-), mais offre une grande souplesse (+) et la possibilité de choisir (+) entre une gestion a priori ou a posteriori.	Les conflits sont gérés implicitement (+), mais suivant une règle imposée (-) a posteriori. De plus, exige la présence de champs d'un type particulier (-).
Portabilité	0.0 2	**2.0** **2**	La portabilité est quasi nulle puisque JDBC permet l'utilisation de SQL propriétaire à la base de données sous-jacente (-).	La portabilité est totale, pourvu que l'on dispose d'un mécanisme de traduction vers le type de base de données souhaité (+).
Performance	2.0 2	1.5 2	La surcharge inutile du client, du réseau et du serveur de base de données causée par l'envoi de requêtes erronées (-) est largement compensé par la puissance offerte par SQL (+).	Cette surcharge a été fort réduite pour améliorer la réactivité du client (+). Par contre, la plus faible complétude du langage impose l'utilisation d'alternatives parfois peu performantes (-).
Répartition	0.0 2	**2.0** **2**	La répartition est d'office possible entre le pilote JDBC et la base de données, mais pas entre le client et le pilote JDBC (-).	Le client peut être distant du moteur (+).

Tableau VII-1 Grille d'évaluation entre JDBC et ARROW

VII.3 Une autre approche

Dans un de ses ouvrages [Heinckiens98], Heinckiens propose un autre type d'architecture, destiné à la manipulation des bases de données suivant des techniques orientées objets. Il s'agit de SCOOP (SCalable Object-Oriented Persistence). A première vue, son objectif rejoint celui que nous avons poursuivi, c'est-à-dire connecter l'univers du développement orienté objet à celui des bases de données, en particulier des bases de données relationnelles. Toutefois, nous allons relever dans cette section les éléments fondamentaux qui permettent aux deux architectures de se distinguer.

VII.3.1 Philosophie

Présentée dans ce travail, l'architecture ARROW permet de créer des modèles orientés objet à partir de schémas de base de données existantes (relationnelles ou autres). On a vu qu'une instance de *WObject* est une représentation objet d'une et une seule entité physique (typiquement une table ou une vue) de la base de données sous-jacente. De même, une instance de *WProperty* est l'équivalent du concept de champ.
C'est donc la structure de la base de données qui conditionne naturellement la modélisation objet. En d'autres termes, nous avons adopté une approche "bottom-up", comme l'illustre la Figure VII-1.

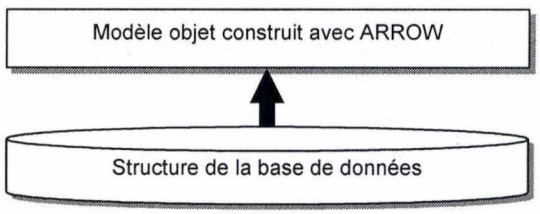

Figure VII-1: L'approche "bottom-up" de ARROW.

A l'inverse, SCOOP propose une approche dite "top-down", comme l'illustre la Figure VII-2. L'objectif n'est plus de reproduire la structure de la base de données sous une forme orientée objet, mais de permettre à des modèles objets d'assurer leur persistence. SCOOP offre la possibilité de mettre en relation un modèle métier orienté objet ("object oriented business data model") et un gestionnaire de persistance quelconque, par exemple une base de données relationnelle.

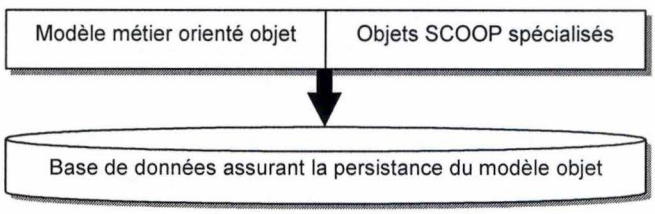

Figure VII-2: L'approche "top-down" de SCOOP.

Des objets spécialisés ("impedance mismatch resolvers") se chargent de la mise en correspondance de la représentation des données entre le modèle objet - dont il faut assurer la persistance - et la base de données. Il est par là possible qu'un même objet gère sa persistance au moyen de plusieurs tables, contrairement au modèle ARROW, où un objet emballe une et une seule table (ou vue).

En résumé, ARROW permet la création d'un modèle objet dont la structure est inspirée de la base de données sous-jacente. SCOOP permet à un modèle objet existant (typiquement un modèle métier) d'assurer sa persistance dans une base de données.

VII.3.2 Architecture

L'architecture SCOOP s'affiche comme une base d'inspiration au développement de modèles métier devant gérer des données persistantes. Ses spécifications se limitent à la définition de trois classes abstraites dont la structure est relativement simple. C'est au développeur de spécialiser ces classes suivant la méthodologie proposée par l'auteur. Comparativement au modèle ARROW, la quantité de code à écrire est importante, bien qu'il soit également envisageable d'utiliser des générateurs.

a) Persistance d'un objet

SCOOP offre un mécanisme de stockage des données membres d'un objet dans une ou plusieurs tables. Par exemple, une classe *Personne* contenant - entre autres - une liste de quatre adresses peut gérer sa persistance soit dans une seule table (qui dispose d'un seul champ d'adresses de cardinalité 4, ou qui contient quatre champs d'adresses distincts), soit dans deux tables reliées entre elles par une relation 1 vers plusieurs, comme l'illustre la Figure VII-3.

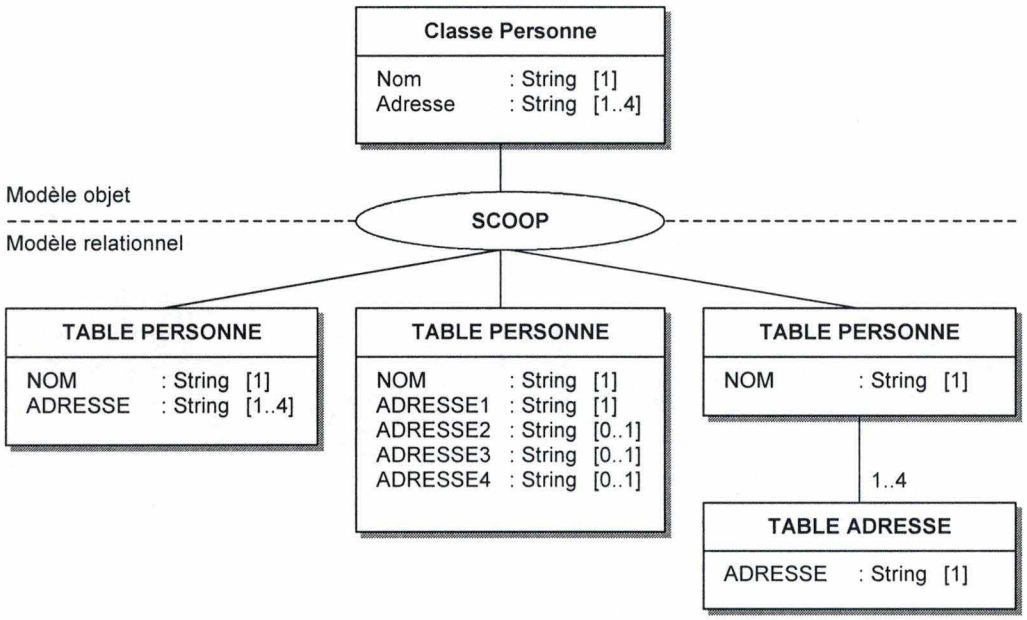

Figure VII-3: La structure des données au niveau du modèle relationnel est rendue transparente au modèle objet.

La mise en concordance des deux modèles est rendue possible grâce aux "impedance mismatch resolvers". Il s'agit d'objets dédiés à la conversion des types de données, comme par exemple le stockage d'une instance classe *Personne* dans deux tables.

Cette abstraction de la base de données - l'utilisateur du modèle objet n'a aucune connaissance de la structure de la base de données sous-jacente - n'est actuellement pas réalisable avec la sémantique proposée par le modèle ARROW, puisqu'à un objet ne peut correspondre plus d'une table. Une solution intermédaire est de composer des vues (si le gestionnaire de la base de données le permet).

b) Opérations sur les objets

SCOOP pose les bases nécessaires à l'exploitation des objets, c'est-à-dire charger, modifier, créer ou supprimer. Les objets peuvent être sélectionnés sur la base de critères simples.
Notons que l'auteur fournit un certain nombre de pistes pour la gestion de la navigation, des requêtes (avec conditions et ordres de tri), des transactions, des événements, de la concurrence et de l'intégrité référentielle, mais il n'a pas incorporé, au modèle SCOOP, de sémantique répondant à ces concepts.

c) En résumé

En réalité, SCOOP est la partie centrale d'une méthodologie intégrant d'autres aspects du développement logiciel allant de la conception de schéma de base de données à la réalisation d'interfaces utilisateur. Il propose au concepteur d'application un schéma d'architecture orienté objet dont l'objectif principal est la construction de modèles métier solides.
Par contre, le modèle ARROW doit plutôt être considéré comme une API (application program interface) dont le rôle se "limite" à proposer des structures orientées objet permettant la manipulation de bases de données.

VIII
Conclusion

L'objectif de ce travail était de poser les fondements d'un modèle orienté objet de manipulation de bases de données, principalement relationnelles. Nous avons construit ce modèle (ARROW) sur la base des reproches imputables aux méthodes orientées objet d'accès aux données relationnelles, dont JDBC était la parfaite illustration.
L'établissement d'une liste de critères d'évaluation nous a permis de cadrer et d'argumenter la problématique. Grâce à une pondération associée à chacun des critères, nous avons pu apprécier combien le modèle JDBC était tantôt proche, tantôt éloigné de nos attentes.

Le modèle ARROW a été proposé en réponse aux défauts de JDBC, sans avoir la prétention de s'afficher comme une solution complète voire idéale. Il permet au développeur de manipuler les bases de données d'une manière orientée objet, sans devoir composer de requêtes SQL, et sans devoir se préoccuper des conversions de types entre l'environnement de développement et la base de données, ... et ce, grâce au langage de manipulation dont nous avons détaillé la sémantique.
La mise à disposition d'un langage de définition (distinct du langage de manipulation) nous a permis de déléguer la tâche de création des classes d'encapsulation à un métier analogue à celui d'administrateur de base de données.
Ces deux métiers (développeur et administrateur) exploitent l'interface que nous avons définie comme étant la couche client de notre modèle. Aucun des deux ne doit se soucier du mécanisme assurant la persistance des objets, lequel est géré par la couche moteur.

La couche moteur est le composant distribuable de notre architecture chargé de traduire les ordres (et les données y afférentes) de la couche client dans un langage compréhensible à la base de données sous-jacente. Pour ce, il dispose d'une interface que la couche client se devra d'utiliser et qui permet d'abstraire totalement le type de base de données avec lequel il dialogue. En conséquence, notre modèle peut également s'afficher comme une solution permettant l'intégration logicielle de bases de données hétérogènes. En effet, rien n'empêche une même instance de couche client de se connecter à plusieurs instances de couche moteur, chacune d'elle étant reliée à un type de base de données différent.
Même si notre modèle fournit "en standard" un composant moteur capable d'exploiter les bases de données relationnelles implémentant le SQL/ANSI, nous avons constaté la nécessité de définir un troisième métier dont le rôle est de créeer de nouveaux composants moteur (voire de spécialiser des composants moteur existants).

Enfin, après avoir présenté les différents composants de ARROW, nous avons procédé à son évaluation à partir des critères que nous avions définis dans le cadre de la critique de JDBC. Nous avons ensuite confronté les deux modèles (ARROW et JDBC) pour en relever les défauts et/ou avantages.
Si ARROW répond aux critiques fondamentales relevées sur JDBC, nous avons néanmoins dû pénaliser l'aspect complétude des langages. En effet, dans ARROW, nous avons perdu la richesse et la puissance du SQL. C'était le prix à payer.
Par contre, l'intégration d'une gestion événementielle, d'une gestion de l'intégrité référentielle ainsi qu'une solution aux autres limites que nous avons posées au cours de ce travail, enrichiraient substantiellement notre modèle et le placeraient dans une positon encore plus favorable vis-à-vis de JDBC.

La présentation succincte d'un autre type d'approche, un modèle destiné à assurer la persistance des objets dans des bases de données quelconques, nous a permis de constater que, d'apparence similaire, deux études peuvent suivre une démarche totalement différente.

Pour certains, ce travail paraîtra plus proche de l'exercice de style que d'une réponse concrète à un problème posé. Pour d'autres, il s'agira d'une base d'inspiration, voire peut-être de LA solution...

Bibliographie

[Blaha98] Michael Blaha and William Premerlani, *Object Oriented Modeling and Design for Database Applications*, Prentice Hall, USA, 1998

[Burleson99] Donald K. Burleson, *Inside the Database Object Model*, CRC, USA, 1999

[Cattell97] Rick G. G. Cattell, *Bases de données orientées objets 2ème édition*, Addison Wesley, Paris, 1997

[Giovinazzo00] Williaw A. Giovinazzo, *Object Oriented Data Warehouse Design: Building a Star Schema*, Prentice Hall, USA, 2000

[Heinckiens98] Peter M.Heinckiens, *Building Scalable Database Applications: Object-Oriented Design, Architectures and Implementations*, Addison Wesley, United Kingdom, 1998

[JavaProg00] S. Allamaraju, K. Avedal, R. Browett, J. Diamond, J. Griffin, M. Holden, A. Hoskinson, R. Johnson, T. Karsjens, L. Kim, A. Longshaw, T. Myers, A. Nakhimovsky, D. O'Connor, S. Tyagi, G. Van Damme, G. Van Huizen, M. Wilcox, S. Zeiger, *Professional Java Server Programming J2EE Edition*, Wrox Press, USA 2000

[JavaSDK13] *Java 2 SDK Version 1.3 Standard Edition Documentation*, Sun Microsystems, 2000, www.sun.com

[Lausen98] Georg Lausen and Gottfried Vossen, *Models and Languages of Object-Oriented Databases*, Addison Wesley, United Kingdom, 1998

[ODMG30] Rick G.G. Cattell and Douglas K. Barry, *The Object Data Standard ODMG 3.0*, Morgan Kaufmann, USA, 2000

[Orfali96] R. Orfali, *Object Persistence Beyond Object-Oriented Database*, Prentice Hall, USA, 1996

[Russell00] Craig Russell, *Java Data Object Draft 0.6*, Sun Microsystems, 2000, www.sun.com

[Stonebracker99] Michael Stonebraker and Paul Brown, *Object-Relational DBMSs: Tracking the Next Great Wave Second Edition*, Morgan Kaufmann, USA, 1999

[Weber99] Joseph L. Weber, *Using Java 2 Platform Special Edition*, QUE, USA, 1999

Annexes

Ce chapitre présente les interfaces et classes publiques du modèle ARROW et de la base de données illustrative. Sauf exception, les méthodes et attributs privés ne sont pas présentés.
Le corps des méthodes de ARROW est substitué par "{...}". Le code source complet est disponible sur demande (*bernard.macours@wanadoo.be*).

Package Java du modèle ARROW

Classes de base (package "arrow")

a) WProperty

```
abstract public class WProperty
{
  public final static int ATTR_KEY      = 1;
  public final static int ATTR_REQUIRED = 2;
  public final static int ATTR_STAMP    = 4;

  protected WProperty(String fieldName) {...}
  protected WProperty(String fieldName, int attrs) {...}

  public          Object getValue() {...}
  public          void   setValue(Object value) {...}
  public abstract void   checkValue(Object value);
  public abstract void   checkClass(Object value);

  public void    setDirty(boolean dirty) {...}
  public boolean isDirty() {...}

  protected void    setParent(WObject object) {...}
  public    WObject getParent() {...}

  public int     getAttrs() {...}
  public boolean isRequired() {...}
  public boolean isKey() {...}
  public boolean isStamp() {...}

  public String getFieldName() {...}
  public void   setLabel(String label) {...}
  public String getLabel() {...}
  public void   setDescription(String description) {...}
  public String getDescription() {...}

  protected void   assignSnapshot() {...}
  protected void   revertSnapshot() {...}
  protected Object getSnapshot() {...}
  protected void   clearSnapshot() {...}

  public void willCreate() {...}
  public void willModify() {...}
  public void willRemove() {...}
```

```
  protected void backup() {...}
  protected void restore() {...}

  protected void    fix() {...}
  protected boolean isFixed() {...}

  public String toString() {...}
}
```

b) WReference

```
abstract public class WReference
{
  public WReference() {...}

  protected void    setParent(WObject object) {...}
  public    WObject getParent() {...}

  protected abstract void       checkTarget(WObject targetObject);
  protected abstract WCondition makeCondition(WObject targetObject,
                                              boolean onValues);

  public WObject     getTarget(WObject targetObject) {...}
  public WCollection getTarget(WCollection targetCollection) {...}
  public WCollection getTarget(WCollection targetCollection,
                               WObject targetObject) {...}

  protected void    fix() {...}
  protected boolean isFixed() {...}
}
```

c) WObject

```
abstract public class WObject extends WTransactionComponentAdapter
{
  public final static int STATUS_NEW      = 0;
  public final static int STATUS_LOADED   = 1;
  public final static int STATUS_CREATED  = 2;
  public final static int STATUS_MODIFIED = 3;
  public final static int STATUS_REMOVED  = 4;
  public final static int STATUS_RESET    = 5;

  protected WObject(WArchiver archiver, String tableName) {...}

  public void load() {...}
  public void load(WCondition condition)  {...}
  public void reload() {...}
  public void create() {...}
  public void modify() {...}
  public void save() {...}
  public void remove() {...}

  public void revert() {...}
  public void setNull() {...}
  public void reset() {...}

  protected void    setStatus(int status) {...}
  public    int     getStatus() {...}
  public    boolean isBinded() {...}
  public    void    setDirty(boolean dirty) {...}
  public    boolean isDirty() {...}

  protected void checkIntegrity() {...}
```

```
  protected void backup() {...}
  protected void restore() {...}

  protected void      registerProperty(WProperty property) {...}
  public    WProperty getProperty(int n) {...}
  public    int       countProperties() {...}

  protected void    registerReference(WReference reference) {...}
  public WReference getReference(int n) {...}
  public int        countReferences() {...}

  protected void register() {...}

  public String getTableName() {...}
  public void   setLabel(String label) {...}
  public String getLabel() {...}
  public void   setDescription(String description) {...}
  public String getDescription() {...}

  public String getAlias() {...}

  protected void fix() {...}
  protected boolean isFixed() {...}

  public void joiningTransaction(WTransaction transaction) {...}
  public void completingTransaction() {...}
  public void abortingTransaction() {...}

  public WArchiver getArchiver() {...}

  public void copyFrom(WObject object) {...}
  public void copyFrom(WObject object, boolean onlyValues) {...}
}
```

d) WCondition

```
abstract public class WCondition
{
  public void    setNegated(boolean negated) {...}
  public boolean isNegated() {...}
}
```

e) WSortOrder

```
public class WSortOrder
{
  public WSortOrder() {...}

  public void addProperty(WProperty property, boolean ascending) {...}
  public void ascending(WProperty property) {...}
  public void descending(WProperty property) {...}

  public WProperty getProperty(int n) {...}
  public boolean   isAscending(int n) {...}

  public int countProperties() {...}
}
```

f) WCollection

```
public class WCollection extends WTransactionComponentAdapter
{
  public WCollection() {...}

  public void registerObject(WObject object) {...}
  public void registerObject(WObject object,
                                          WReference joinReference) {...}
  public WObject getObject(int n) {...}
  public int     countObjects() {...}

  public void       setCondition(WCondition condition) {...}
  public WCondition getCondition() {...}

  public void       setSortOrder(WSortOrder sortOrder) {...}
  public WSortOrder getSortOrder() {...}


  public void    refresh() {...}
  public boolean next() {...} {...}
  public int     count() {...}
  public void    setMaxNexts(int maxNexts) {...}
  public int     getMaxNexts() {...}
  public void    modify(WObject object) {...}
  public void    remove(WObject object) {...}

  public int     getPosition() {...}
  public boolean isBeforeFirst() {...}
  public boolean isFirst() {...}
  public boolean isLast() {...}
  public boolean isAfterLast() {...}

  public void joiningTransaction(WTransaction transaction) {...}
  public void completingTransaction() {...}
  public void abortingTransaction() {...}

  public WArchiver getArchiver() {...}

  public String getLabel() {...}
  public void   setLabel(String sLabel) {...}

  public String getDescription() {...}
  public void   setDescription(String sDescription) {...}
}
```

g) WTransaction

```
public class WTransaction
{
  public final static int STATUS_NEW         = 0;
  public final static int STATUS_ABORTING    = 1;
  public final static int STATUS_ABORTED     = 2;
  public final static int STATUS_COMPLETING  = 3;
  public final static int STATUS_COMPLETED   = 4;
  public final static int STATUS_CLEANINGUP  = 5;
  public final static int STATUS_CLEANUP     = 6;

  public WTransaction() {...}

  public void    registerComponent(WTransactionComponent component){...}
  public boolean containsComponent(WTransactionComponent component){...}
  public WTransactionComponent getComponent(int n) {...}
  public int     countComponents() {...}
```

```
  public void complete() {...}
  public void abort() {...}
  public void cleanup() {...}

  public int getStatus() {...}

  public boolean isNew() {...}
  public boolean isAborted() {...}
  public boolean isCompleted() {...}
  public boolean isCleanup() {...}
  public boolean isStartedInArchiver() {...}

  protected void finalize() {...}

  public WArchiver getArchiver() {...}

  public String getAlias() {...}
}
```

h) WTransactionComponent

```
public interface WTransactionComponent
{
  public void         joiningTransaction(WTransaction transaction);
  public void         leavingTransaction();
  public WTransaction getTransaction();

  public void         willCompleteTransaction();
  public void         completingTransaction();
  public void         willAbortTransaction();
  public void         abortingTransaction();

  public WArchiver    getArchiver();
}
```

i) WArchiver

```
public interface WArchiver
{
  abstract public WObject load(WObject object, WCondition condition);
  abstract public int     count(WObject[] objects, WCondition condition);

  abstract public long    create(WObject    object,
                                 String     transactionAlias);
  abstract public long    modify(WObject    object,
                                 WCondition condition,
                                 String     transactionAlias);
  abstract public long    remove(WObject    object,
                                 WCondition condition,
                                 String     transactionAlias);

  abstract public void    complete(String transactionAlias);
  abstract public void    abort(String transactionAlias);

  abstract public WIterator createIterator(WObject[]  objects,
                                           WCondition condition,
                                           WSortOrder sortOrder,
                                           int        maxNexts,
                                           boolean    positionnable);

  abstract public boolean isStarted(String transactionAlias);
  abstract public void    cleanup();
}
```

j) WIterator

```
public interface WIterator
{
  public WObject[] next();
  public boolean   hasNext();
  public void      setPosition(int position);
  public int       getPosition();
  public void      cleanup();
  public WArchiver getArchiver();
}
```

Spécialisations de WProperty (package "arrow.property")

a) WAnyProperty

```
public class WAnyProperty extends WProperty
{
  public WAnyProperty(String fieldName, String className) {...}
  public WAnyProperty(String fieldName, int attrs, String className){...}

  public void checkClass(Object value) {...}
  public void checkValue(Object value) {...}

  public void   set(Object o) {...}
  public Object get() {...}
}
```

b) WDateProperty

```
public class WDateProperty extends WProperty
{
  public WDateProperty(String fieldName) {...}
  public WDateProperty(String fieldName, int attrs) {...}

  public void checkClass(Object value) {...}
  public void checkValue(Object value) {...}

  public void           set(java.util.Date date) {...}
  public java.util.Date get() {...}
}
```

c) WDoubleProperty

```
public class WDoubleProperty extends WProperty
{
  public WDoubleProperty(String fieldName) {...}
  public WDoubleProperty(String fieldName, int attrs) {...}

  public void checkClass(Object value) {...}
  public void checkValue(Object value) {...}

  public void   set(double d) {...}
  public double get() {...}
}
```

d) WIntegerProperty, WIntervalIntegerProperty et WVersionProperty

```
public class WIntegerProperty extends WProperty
{
  public WIntegerProperty(String fieldName) {...}
  public WIntegerProperty(String fieldName, int attrs) {...}

  public void checkClass(Object value) {...}
  public void checkValue(Object value) {...}

  public void set(int n) {...}
  public int  get() {...}
}
```

```
public class WIntervalIntegerProperty extends WIntegerProperty
{
  public WIntervalIntegerProperty(String fieldName,
                                  int    min,
                                  int    max) {...}
  public WIntervalIntegerProperty(String fieldName,
                                  int    attrs,
                                  int    min,
                                  int    max) {...}

  public void checkValue(Object value) {...}

  public int getMin() {...}
  public int getMax() {...}
}
```

```
public class WVersionProperty extends WIntegerProperty
{
  public WVersionProperty(String fieldName) {...}
  public WVersionProperty(String fieldName, int attrs) {...}

  public void willCreate() {...}
  public void willModify() {...}
}
```

e) WStringProperty, WLimitedStringProperty, WIdProperty et WForeignIdProperty

```
public class WStringProperty extends WProperty
{
  public WStringProperty(String fieldName) {...}
  public WStringProperty(String fieldName, int attrs) {...}

  public void checkClass(Object value) {...}
  public void checkValue(Object value) {...}

  public void   set(String s) {...}
  public String get() {...}
}
```

```
public class WLimitedStringProperty extends WStringProperty
{
  public WLimitedStringProperty(String fieldName,
                                int    min,
                                int    max) {...}
  public WLimitedStringProperty(String fieldName,
                                int    attrs,
                                int    min,
                                int    max) {...}
```

```
  public void checkClass(Object value) {...}
  public void checkValue(Object value) {...}

  public int getMin() {...}
  public int getMax() {...}
}
```

```
public class WIdProperty extends WLimitedStringProperty
{
  public WIdProperty(String fieldName, int length) {...}
  public WIdProperty(String fieldName, int attrs, int length) {...}

  public void willCreate() {...}

  public String makeId() {...}
}
```

```
public class WForeignIdProperty extends WLimitedStringProperty
{
  public WForeignIdProperty(String fieldName, int length) {...}
  public WForeignIdProperty(String fieldName, int attrs, int length){...}
}
```

Spécialisations de WReference (package "arrow.reference")

```
public class WAnyReference extends WReference
{
  public WAnyReference(String targetClassName) {...}
  public WAnyReference(String    targetClassName,
                       WProperty sourceProperty,
                       String    targetPropertyName) {...}

  protected void checkTarget(WObject targetObject) {...}
  protected WCondition makeCondition(WObject targetObject,
                                     boolean valuesElseProperties) {...}

  public void addLink(WProperty sourceProperty,
                      String    targetPropertyName) {...}

  public int countLinks() {...}

  public WProperty getSourceProperty(int n) {...}
  public String    getTargetPropertyName(int n) {...}
  public String    getTargetClassName() {...}
}
```

Spécialisations de WCondition (package "arrow.condition")

a) WSimpleCondition

```
public class WComplexCondition extends WCondition
{
  public WComplexCondition() {...}

  public void addCondition(WCondition oCondition, boolean bOr) {...}
  public void and(WCondition oCondition) {...}
  public void or(WCondition oCondition) {...}

  public WCondition getCondition(int nIndex) {...}
  public boolean    isAnd(int nIndex) {...}
  public boolean    isOr(int nIndex) {...}
```

```
  public int countConditions() {...}
}
```

b) WSimpleCondition

```
abstract public class WSimpleCondition extends WCondition
{
  public WSimpleCondition(WProperty property) {...}

  public WProperty getProperty() {...}
}
```

c) WSimpleValueCondition et spécialisations (W...ValueCondition)

```
abstract public class WSimpleValueCondition extends WSimpleCondition
{
  public WSimpleValueCondition(WProperty property) {...}
  public WSimpleValueCondition(WProperty property, Object value) {...}

  public void   setValue(Object value) {...}
  public Object getValue() {...}
}
```

```
public class WIsEqualToValue extends WSimpleValueCondition
{
  public WIsEqualToValue(WProperty property) {...}
  public WIsEqualToValue(WProperty property, boolean negated) {...}
  public WIsEqualToValue(WProperty property, Object value) {...}
  public WIsEqualToValue(WProperty property, Object value,
                                                boolean negated) {...}

  public void   addValue(Object value) {...}
  public Object getValue(int index) {...}
  public int    countValues() {...}
}
```

```
public class WIsGreaterThanValue extends WSimpleValueCondition
{
  public WIsGreaterThanValue(WProperty property) {...}
  public WIsGreaterThanValue(WProperty property, boolean strict) {...}
  public WIsGreaterThanValue(WProperty property, Object value) {...}
  public WIsGreaterThanValue(WProperty property, Object value,
                                                   boolean strict) {...}
  public void    setStrict(boolean strict) {...}
  public boolean isStrict() {...}
}
```

```
public class WIsLessThanValue extends WSimpleValueCondition
{
  public WIsLessThanValue(WProperty property) {...}
  public WIsLessThanValue(WProperty property, boolean strict) {...}
  public WIsLessThanValue(WProperty property, Object value) {...}
  public WIsLessThanValue(WProperty property, Object value,
                                                boolean strict) {...}

  public void    setStrict(boolean strict) {...}
  public boolean isStrict() {...}
}
```

```
public class WMatchesValue extends WSimpleValueCondition
{
  public WMatchesValue(WProperty property) {...}
  public WMatchesValue(WProperty property, Object value) {...}
}
```

d) WSimplePropertyCondition et spécialisations (W...PropertyCondition)

```
abstract public class WSimplePropertyCondition extends WSimpleCondition
{
  public WSimplePropertyCondition(WProperty property,
                                  WProperty targetProperty) {...}

  public void      setTargetProperty(WProperty targetProperty) {...}
  public WProperty getTargetProperty() {...}
}
```

```
public class WIsEqualToProperty extends WSimplePropertyCondition
{
  public WIsEqualToProperty(WProperty property,
                            WProperty targetProperty) {...}
}
```

```
public class WIsGreaterThanProperty extends WSimplePropertyCondition
{
  public WIsGreaterThanProperty(WProperty property,
                                WProperty targetProperty) {...}
  public WIsGreaterThanProperty(WProperty property,
                                WProperty targetProperty,
                                boolean   strict) {...}

  public void    setStrict(boolean strict) {...}
  public boolean isStrict() {...}
}
```

```
public class WIsLessThanProperty extends WSimplePropertyCondition
{
  public WIsLessThanProperty(WProperty property,
                             WProperty targetProperty) {...}
  public WIsLessThanProperty(WProperty property,
                             WProperty targetProperty,
                             boolean   strict) {...}

  public void    setStrict(boolean strict) {...}
  public boolean isStrict() {...}
}
```

```
public class WMatchesProperty extends WSimplePropertyCondition
{
  public WMatchesProperty(WProperty property,
                          WProperty targetProperty) {...}
}
```

Moteur JDBC/SQL (package "arrow.sql")

a) WJdbcArchiver et WSunOdbcJdbcArchiver

```
abstract public class WJdbcArchiver implements WArchiver
{
  static private WSessionPool pool = new WSessionPool();

  public WObject load(WObject object, WCondition condition) {...}
  public int     count(WObject[] objects, WCondition condition) {...}

  public long create(WObject    object,
                     String     transactionAlias) {...}
  public long modify(WObject    object,
                     WCondition condition,
                     String     transactionAlias) {...}
  public long remove(WObject    object,
                     WCondition condition,
                     String     transactionAlias) {...}

  public void    complete(String transactionAlias) {...}
  public void    abort(String transactionAlias) {...}
  public boolean isStarted(String transactionAlias) {...}

  abstract protected WSqlTranslator getSqlTranslator();
  abstract protected Connection     createConnection();

  public WIterator createIterator(WObject[]  objects,
                                  WCondition condition,
                                  WSortOrder sortOrder,
                                  int        maxNexts,
                                  boolean    positionnable) {...}

  synchronized public void cleanup() {...}
  protected           void finalize() {...}
}
```

```
public class WSunOdbcJdbcArchiver extends WJdbcArchiver
{
  private WSqlTranslator sqlTranslator = new WAnsiSqlTranslator();

  public WSunOdbcJdbcArchiver(String url) {...}

  public WSunOdbcJdbcArchiver(String url,
                              String user,
                              String password) {...}

  protected Connection createConnection() {...}

  protected WSqlTranslator getSqlTranslator() {...}
}
```

b) WJdbcIterator

```
public class WJdbcIterator implements WIterator
{
  protected WJdbcIterator(WJdbcArchiver archiver,
                          WObject[]     objects,
                          WCondition    condition,
                          WSortOrder    sortOrder,
                          int           maxNexts,
                          boolean       positionnable) {...}

  public WObject[] next() {...}
```

```
  public boolean   hasNext() {...}
  public void      setPosition(int position) {...}
  public int       getPosition() {...}

  public WArchiver getArchiver() {...}

  public void cleanup() {...}
}
```

c) WSqlTranslator et WAnsiSqlTranslator

```
public interface WSqlTranslator
{
  public String getSqlLoad   (WObject    object,
                              WCondition condition,
                              Vector     params);
  public String getSqlLoad   (WObject[]  objects,
                              WCondition condition,
                              WSortOrder sortOrder,
                              Vector     params);
  public String getSqlCount  (WObject[]  objects,
                              WCondition condition,
                              Vector     params);
  public String getSqlCreate (WObject    object,
                              Vector     params);
  public String getSqlModify (WObject    object,
                              WCondition condition,
                              Vector     params);
  public String getSqlRemove (WObject    object,
                              WCondition condition,
                              Vector     params);
}
```

```
public class WAnsiSqlTranslator implements WSqlTranslator
{
  public String getSqlLoad   (WObject    object,
                              WCondition condition,
                              Vector     params) {...}
  public String getSqlLoad   (WObject[]  objects,
                              WCondition condition,
                              WSortOrder sortOrder,
                              Vector     params) {...}
  public String getSqlCount  (WObject[]  objects,
                              WCondition condition,
                              Vector     params) {...}
  public String getSqlCreate (WObject    object,
                              Vector     params) {...}
  public String getSqlModify (WObject    object,
                              WCondition condition,
                              Vector     params) {...}
  public String getSqlRemove (WObject    object,
                              WCondition condition,
                              Vector     params) {...}
}
```

d) WSession et WSessionPool

```
public class WSession
{
  protected WSession(WArchiver  archiver,
                     String     transactionAlias,
                     Connection connection) {...}
```

```
  synchronized protected void reserve(WArchiver archiver,
                                      String    transactionAlias) {...}

  synchronized public void release() {...}

  public boolean    isReserved() {...}
  public WArchiver  getArchiver() {...}
  public String     getTransactionAlias() {...}
  public Connection getConnection() {...}

  protected long getTimeSinceLastLock() {...}
  protected long getTimeSinceLastUnlock() {...}
}
```

```
public class WSessionPool
{
  public WSessionPool() {...}

  synchronized public WSession register(WArchiver  archiver,
                                        String     transactionAlias,
                                        Connection connection) {...}

  synchronized public WSession reserve(WArchiver archiver,
                                       String    transactionAlias) {...}

  synchronized public WSession find(WArchiver archiver,
                                    String    transactionAlias) {...}

  synchronized public void release(WArchiver archiver) {...}
}
```

Package Java de la base de données illustrative

Définition de la base de données (package "arrowx.database")

a) Spécialisations de WProperty

```
public class XCodeProperty extends WLimitedStringProperty
{
  public XCodeProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_REQUIRED, 0, 10); }
}
```

```
public class XDateProperty extends WDateProperty
{
  public XDateProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0); }
}
```

```
public class XForeignIdProperty extends WForeignIdProperty
{
  public XForeignIdProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0, 16); }
}
```

```
public class XIdProperty extends WIdProperty
{
  public XIdProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_KEY | WProperty.ATTR_REQUIRED, 16); }
}
```

```
public class XNomProperty extends WLimitedStringProperty
{
  public XNomProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_REQUIRED, 0, 50); }
}
```

```
public class XPrixProperty extends WDoubleProperty
{
  public XPrixProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0); }
}
```

```
public class XQuantitéProperty extends WIntegerProperty
{
  public XQuantitéProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0); }
}
```

```
public class XRayonProperty extends WIntervalIntegerProperty
{
  public XRayonProperty(String fieldName, boolean required)
  { super(fieldName, required ? WProperty.ATTR_REQUIRED : 0, 1, 31); }
}
```

```
public class XVersionProperty extends WVersionProperty
{
  public XVersionProperty(String fieldName)
  { super(fieldName, WProperty.ATTR_REQUIRED | WProperty.ATTR_STAMP); }
}
```

b) Spécialisations de WObject

```
public abstract class XObject extends WObject
{
  static private WArchiver archiver = new
                                  WSunOdbcJdbcArchiver("jdbc:odbc:Arrow");

  final public XIdProperty      id      = new XIdProperty("ID");
  final public XVersionProperty version = new XVersionProperty("VERSION");

  public XObject(String tableName)
  { super(archiver, tableName); }
}
```

```
final public class XClient extends XObject
{
  public XCodeProperty code = new XCodeProperty("CODE");
  public XNomProperty  nom  = new XNomProperty("NOM");

  public WAnyReference commandesRef = new
              WAnyReference("arrowx.database.XCommande", id, "clientId");

  public XClient()
  {
    super("CLIENT");
    register();
    fix();
  }
}
```

```
final public class XCommande extends XObject
{
  final public XForeignIdProperty clientId = new
                                    XForeignIdProperty("CLIENT_ID", true);
  final public XDateProperty      dateCommande = new
                                     XDateProperty("DATE_COMMANDE", true);
  final public XDateProperty      dateLivraison = new
                                    XDateProperty("DATE_LIVRAISON", true);

  final public WAnyReference clientRef = new
                 WAnyReference("arrowx.database.XClient", clientId, "id");
  final public WAnyReference détailsRef = new
               WAnyReference("arrowx.database.XDétail", id, "commandeId");

  public XCommande()
  {
    super("COMMANDE");
    register();
    fix();
  }

  protected void checkIntegrity()
  {
    super.checkIntegrity();

    if (dateLivraison.get() != null &&
        dateLivraison.get().compareTo(dateCommande.get()) < 0)
      throw (new WException(0));
  }
}
```

```
final public class XDétail extends XObject
{
  final public XForeignIdProperty commandeId = new
                                  XForeignIdProperty("COMMANDE_ID", true);
  final public XForeignIdProperty produitId = new
                                   XForeignIdProperty("PRODUIT_ID", true);
  final public XQuantitéProperty  quantité = new
                                     XQuantitéProperty("QUANTITE", true);
  final public XPrixProperty      prix = new
                                           XPrixProperty("PRIX", true);

  final public WAnyReference commandeRef = new
             WAnyReference("arrowx.database.XCommande", commandeId, "id");
  final public WAnyReference produitRef = new
               WAnyReference("arrowx.database.XProduit", produitId, "id");

  public XDétail()
  {
    super("DETAIL");
    register();
    fix();
  }
}
```

```
final public class XProduit extends XObject
{
  final public XCodeProperty     code  = new XCodeProperty("CODE");
  final public XNomProperty      nom   = new XNomProperty("NOM");
  final public XQuantitéProperty stock = new XQuantitéProperty("STOCK",
                                                                 true);
  final public XPrixProperty     prix  = new XPrixProperty("PRIX", true);
  final public XRayonProperty    rayon = new XRayonProperty("RAYON",
                                                               false);

  final public WAnyReference détailsRef = new
               WAnyReference("arrowx.database.XProduit", id, "produitId");
```

```
  public XProduit()
  {
    super("PRODUIT");
    register();
    fix();
  }
}
```